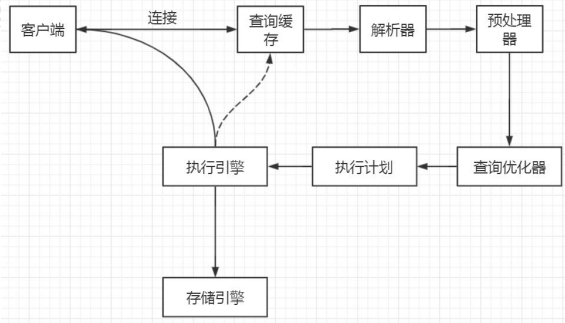
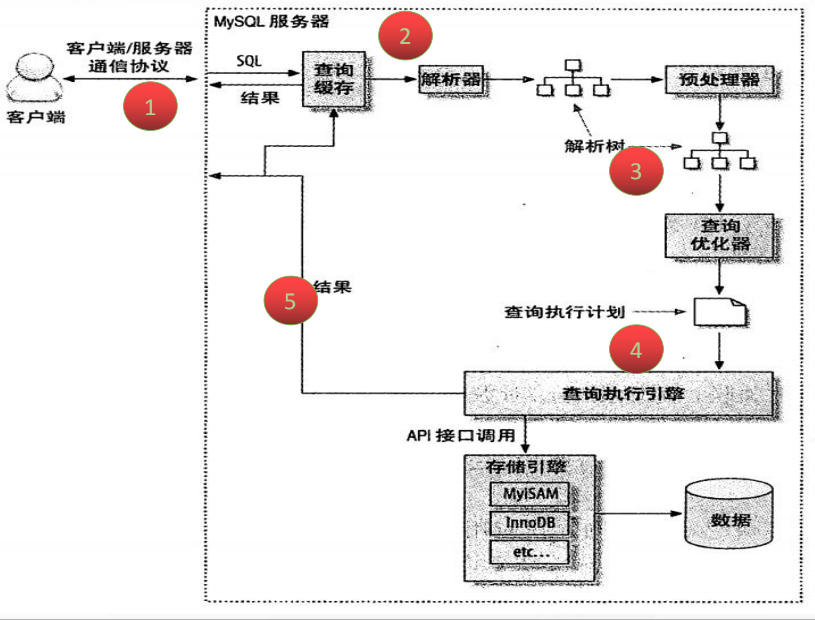


# Mysql架构和执行流程

## 一条查询查询sql的执行过程





### 跟数据库建立连接

#### 通信协议

MySQL 是支持多种通信协议的，可以使用同步/异步的方式，支持长连接/短连接。

通信类型：同步或者异步

同步通信的特点：

1、同步通信依赖于被调用方，受限于被调用方的性能。也就是说，应用操作数据库，线程会阻塞，等待数据库的返回。

2、一般只能做到一对一，很难做到一对多的通信

异步通信的特点

1、异步可以避免应用阻塞等待，但是不能节省 SQL 执行的时间。

2、如果异步存在并发，每一个 SQL 的执行都要单独建立一个连接，避免数据混乱。但是这样会给服务端带来巨大的压力（一个连接就会创建一个线程，线程间切换会占用大量 CPU 资源）。另外异步通信还带来了编码的复杂度，所以一般不建议使用。如果要异步，必须使用连接池，排队从连接池获取连接而不是创建新连接。

**一般来说我们连接数据库都是同步连接**

连接方式

1. 长连接：可以保持打开的连接，减少服务端和释放连接的消耗，后面程序访问的时候还能够使用这个连接；

**一般我们会在连接池中使用长连接**。

保持长连接会消耗内存。长时间不活动的连接，MySQL 服务器会断开

show global variables like 'wait\_timeout'; -- 非交互式超时时间，如 JDBC 程序

show global variables like 'interactive\_timeout'; -- 交互式超时时间，如数据库工具

默认都是 28800 秒，8 小时。

**查看连接的个数**

用show status命令

show global status like 'Thread%';

Threads\_cached：缓存中的线程连接数。

Threads\_connected：当前打开的连接数。

Threads\_created：为处理连接创建的线程数。

Threads\_running：非睡眠状态的连接数，通常指并发连接数。

每产生一个连接或者一个会话，在服务端就会创建一个线程来处理。反过来，如果要杀死会话，就是 Kill 线程

**查看连接状态**

命令show processlist

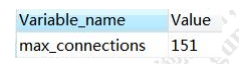
常见的连接状态



MySQL 服务允许的最大连接数是多少呢？

在 5.7 版本中默认是 151 个，最大可以设置成 16384

show variables like 'max\_connections';



1. 短连接：操作完毕之后就马上关闭连接

**通信协议**

第一种是 Unix Socket：在 Linux 服务器上，如果没有指定-h 参数，它就用 socket 方式登录

第二种方式，TCP/IP 协议：如果指定-h 参数，就会用这种方式；我们的编程语言的连接模块都是用 TCP 协议连接到 MySQL 服务器的，比如mysql-connector-java-x.x.xx.jar。

另外还有命名管道（Named Pipes）和内存共享（Share Memory）的方式，这两种通信方式只能在 Windows 上面使用，一般用得比较少。

#### 通信方式

单工：在两台计算机通信的时候，数据的传输是单向的。生活中的类比：遥控器。

半双工：在两台计算机之间，数据传输是双向的，你可以给我发送，我也可以给你发送，但是在这个通讯连接里面，同一时间只能有一台服务器在发送数据，也就是你要给我发的话，也必须等我发给你完了之后才能给我发。生活中的类比：对讲机。

全双工：数据的传输是双向的，并且可以同时传输。生活中的类比：打电话。

**MySQL 使用了半双工的通信方式**

要么是客户端向服务端发送数据，要么是服务端向客户端发送数据，这两个动作不能同时发生。所以客户端发送 SQL 语句给服务端的时候，（在一次连接里面）数据是不能分成小块发送的，不管你的 SQL 语句有多大，都是一次性发送。

比如我们用 MyBatis 动态 SQL 生成了一个批量插入的语句，插入 10 万条数据，values后面跟了一长串的内容，或者 where 条件 in 里面的值太多，会出现问题。这个时候我们必须要调整 MySQL 服务器配置 max\_allowed\_packet 参数的值（默认是 4M），把它调大，否则就会报错

另一方面，对于服务端来说，也是一次性发送所有的数据，不能因为你已经取到了想要的数据就中断操作，这个时候会对网络和内存产生大量消耗。所以，我们一定要在程序里面避免不带 limit 的这种操作，比如一次把所有满足条件的数据全部查出来，一定要先 count 一下。如果数据量的话，可以分批查询。

### 查询缓存

客户端和服务端建立连接后下一步要做的就是查询缓存

缓存从mysql的5.7版本开始就是默认关闭的（在 MySQL 8.0 中，查询缓存已经被移除了）

缓存默认关闭就是不推荐使用，为什么不推荐使用

主要是因为 MySQL 自带的缓存的应用场景有限，第一个是它要求 SQL 语句必须一模一样，中间多一个空格，字母大小写不同都被认为是不同的的 SQL。

第二个是表里面任何一条数据发生变化的时候，这张表所有缓存都会失效，所以对于有大量数据更新的应用，也不适合。

所以缓存这一块，我们还是交给 ORM 框架（比如 MyBatis 默认开启了一级缓存），或者独立的缓存服务，比如 Redis 来处理更合适。

### 语法解析和预处理

Mysql为什么能够判断sql的正确性，sql语法是否正确，表字段是否存在；完成这项工作那么就需要mysql的Parser解析器和Preprocessor 预处理模块

**词法解析**

词法分析就是把一个完整的 SQL 语句打碎成一个个的单词。

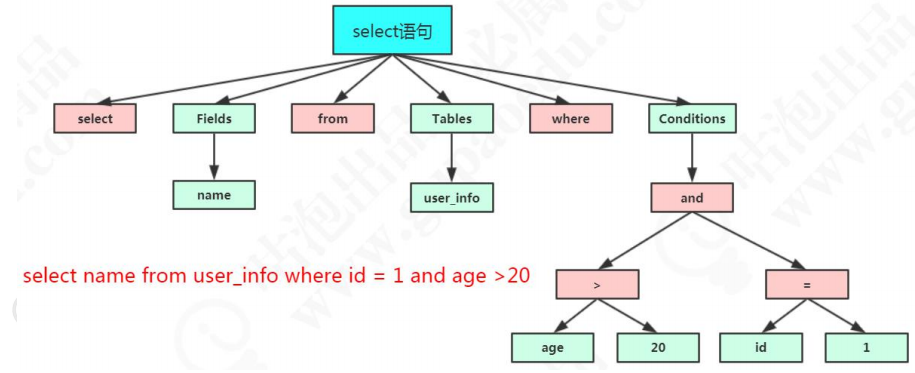
比如一个简单的 SQL 语句：

select name from user where id = 1;

它会打碎成 8 个符号，每个符号是什么类型，从哪里开始到哪里结束

**语法解析**

第二步就是语法分析，语法分析会对 SQL 做一些语法检查，比如单引号有没有闭合，然后根据 MySQL 定义的语法规则，根据 SQL 语句生成一个数据结构。这个数据结构我们把它叫做解析树（select\_lex）。



**预处理器**

如果我写了一个词法和语法都正确的 SQL，但是表名或者字段不存在，会在哪里报错

解析器可以分析语法，但是它怎么知道数据库里面有什么表，表里面有什么字段呢？

实际上还是在解析的时候报错，解析 SQL 的环节里面有个预处理器。它会检查生成的解析树，解决解析器无法解析的语义。比如，它会检查表和列名是否存在，检查名字和别名，保证没有歧义。

预处理之后得到一个新的解析树

### 查询优化和查询执行计划

问题：一条SQL 语句是不是只有一种执行方式？或者说数据库最终执行的 SQL 是不是就是我们发送的 SQL

这个答案是否定的。一条 SQL 语句是可以有很多种执行方式的，最终返回相同的结果，他们是等价的。但是如果有这么多种执行方式，这些执行方式怎么得到的？

最终选择哪一种去执行？根据什么判断标准去选择？这个就是 MySQL 的查询优化器的模块（Optimizer）。

**查询优化器**的目的就是根据解析树生成不同的执行计划（Execution Plan），然后选择一种最优的执行计划，MySQL 里面使用的是基于开销（cost）的优化器，那种执行计划开销最小，就用哪种。

**优化器的可以做什么**

举两个简单的例子：

1、当我们对多张表进行关联查询的时候，以哪个表的数据作为基准表。

2、有多个索引可以使用的时候，选择哪个索引。

实际上，对于每一种数据库来说，优化器的模块都是必不可少的，他们通过复杂的算法实现尽可能优化查询效率的目标。

但是优化器也不是万能的，并不是再垃圾的 SQL 语句都能自动优化，也不是每次都能选择到最优的执行计划，在编写 SQL 语句的时候还是要注意

**优化器得到执行计划的过程**

1. 开启优化器追踪（默认是关闭的）
2. 执行sql,优化器会生成执行计划；优化器的分析过程就会记录到系统表里面

查看优化器记录

select \* from information\_schema.optimizer\_trace\G

它是一个 JSON 类型的数据，主要分成三部分，准备阶段、优化阶段和执行阶段。



expanded\_query 是优化后的 SQL 语句。

considered\_execution\_plans 里面列出了所有的执行计划。

分析完关掉它：

set optimizer\_trace="enabled=off";

SHOW VARIABLES LIKE 'optimizer\_trace';

优化器最终会把解析树变成一个查询执行计划，查询执行计划是一个数据结构。

当然，这个执行计划是不是一定是最优的执行计划呢？不一定，因为 MySQL 也有可能覆盖不到所有的执行计划。

怎么查看 MySQL 的执行计划呢？

在 SQL 语句前面加上 EXPLAIN，就可以看到执行计划的信息

EXPLAIN select name from user where id=1;

\*注意 Explain 的结果也不一定最终执行的方式。

### 存储引擎

Mysql中数据的存放

在关系型数据库里面，数据是放在什么结构里面的？（放在表 Table 里面的）我们可以把这个表理解成 Excel 电子表格的形式。所以我们的表在存储数据的同时，还要组织数据的存储结构，这个存储结构就是由我们的存储引擎决定的，所以我们也可以把存储引擎叫做表类型。

在 MySQL 里面，支持多种存储引擎，他们是可以替换的，所以叫做插件式的存储引擎。

在 MySQL 里面，我们创建的每一张表都可以指定它的存储引擎，而不是一个数据库

只能使用一个存储引擎。存储引擎的使用是以表为单位的。而且，创建表之后还可以修改存储引擎。

我们说一张表使用的存储引擎决定我们存储数据的结构，那在服务器上它们是怎么存储的呢？我们先要找到数据库存放数据的路径：

show variables like 'datadir';

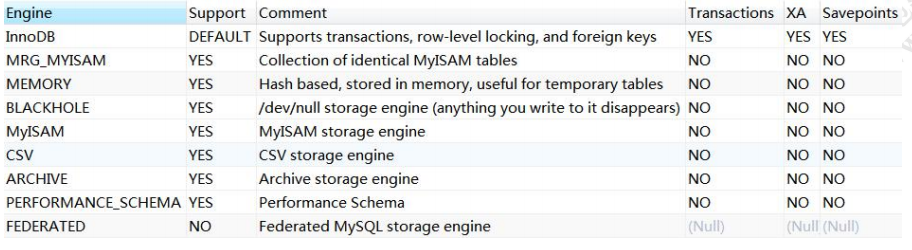
每个存储引擎都有一个**frm文件**，这个就是表结构的定义文件；不同的存储引擎存放数据的方式也不一样，比如InnoDB是一个文件存放数据，myisam使用2个文件存放数据

#### 常见的存储引擎

查看数据库存储引擎的命令：show engines

其中有存储引擎的描述和对事务、XA 协议和 Savepoints 的支持。XA 协议用来实现分布式事务（分为本地资源管理器，事务管理器）。

Savepoints 用来实现子事务（嵌套事务）。创建了一个 Savepoints 之后，事务就可以回滚到这个点，不会影响到创建 Savepoints 之前的操作。



**InnoDB**

mysql 5.7 中的默认存储引擎。InnoDB 是一个事务安全（与 ACID 兼容）的 MySQL

存储引擎，它具有提交、回滚和崩溃恢复功能来保护用户数据。

InnoDB 行级锁（不升级为更粗粒度的锁）和 Oracle 风格的一致非锁读提高了多用户并发性和性能。InnoDB 将用户数据存储在聚集索引中，以减少基于主键的常见查询的 I/O。为了保持数据完整性，

InnoDB 还支持外键引用完整性约束。

特点：

支持事务，支持外键，因此数据的完整性、一致性更高。

支持行级别的锁和表级别的锁。

支持读写并发，写不阻塞读（MVCC）。

特殊的索引存放方式，可以减少 IO，提升查询效率。

适合：经常更新的表，存在并发读写或者有事务处理的业务系统

**Myisam**

应用范围比较小。表级锁定限制了读/写的性能，因此在 Web 和数据仓库配置中，它通常用于只读或以读为主的工作。

特点：

支持表级别的锁（插入和更新会锁表）。不支持事务。

拥有较高的插入（insert）和查询（select）速度。

存储了表的行数（count 速度更快）。（怎么快速向数据库插入 100 万条数据？我们有一种先用 MyISAM 插入数据，然后

修改存储引擎为 InnoDB 的操作。）

适合：只读之类的数据分析的项目

Memory

将所有数据存储在 RAM 中，以便在需要快速查找非关键数据的环境中快速访问。这个引擎以前被称为堆引擎。其使用案例正在减少；InnoDB 及其缓冲池内存区域提供了一种通用、持久的方法来将大部分或所有数据保存在内存中，而 ndbcluster 为大型分布式数据集提供了快速的键值查找。

特点：

把数据放在内存里面，读写的速度很快，但是数据库重启或者崩溃，数据会全部消失。只适合做临时表。将表中的数据存储到内存中。

Csv

它的表实际上是带有逗号分隔值的文本文件。csv表允许以csv格式导入或转储数据，以便与读写相同格式的脚本和应用程序交换数据。因为 csv 表没有索引，所以通常在正常操作期间将数据保存在 innodb 表中，并且只在导入或导出阶段使用 csv 表。

特点：不允许空行，不支持索引。格式通用，可以直接编辑，适合在不同数据库之间导入导出

Archive

用于存储和检索大量很少引用的历史、存档或安全审计信息。

特点：不支持索引，不支持 update delete。

怎么选择存储引擎

1. 如果对数据一致性要求比较高，需要事务支持，可以选择 InnoDB。
2. 如果数据查询多更新少，对查询性能要求比较高，可以选择 MyISAM。
3. 如果需要一个用于查询的临时表，可以选择 Memory。
4. 如果所有的存储引擎都不能满足你的需求，并且技术能力足够，可以根据官网内部

#### 执行引擎

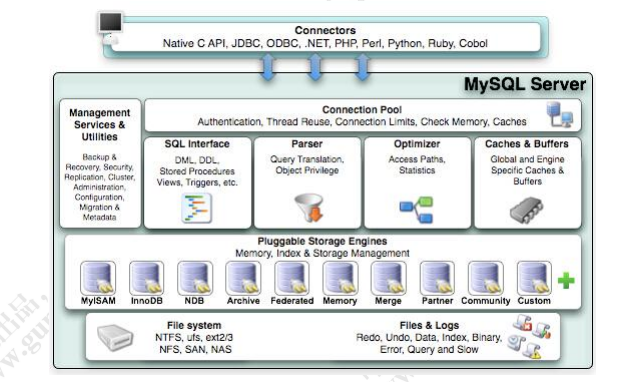
执行引擎使用执行计划去操作存储引擎，它利用存储引擎提供的相应api来完成操作。

为什么我们修改了表的存储引擎，操作方式不需要做任何改变？

因为不同功能的存储引擎实现的 API 是相同的。

最后把数据返回给客户端，即使没有结果也要返回。

## Mysql的体系架构



### 模块详解

1、 Connector：用来支持各种语言和 SQL 的交互，比如 PHP，Python，Java 的JDBC；

2、 Management Serveices & Utilities：系统管理和控制工具，包括备份恢复MySQL 复制、集群等等；

3、 Connection Pool：连接池，管理需要缓冲的资源，包括用户密码权限线程等等；

4、 SQL Interface：用来接收用户的 SQL 命令，返回用户需要的查询结果

5、 Parser：用来解析 SQL 语句；

6、 Optimizer：查询优化器；

7、 Cache and Buffer：查询缓存，除了行记录的缓存之外，还有表缓存，Key 缓存，权限缓存等等；

8、 Pluggable Storage Engines：插件式存储引擎，它提供 API 给服务层使用，跟具体的文件打交道

### 架构分层

**连接层**

我们的客户端要连接到 MySQL 服务器 3306 端口，必须要跟服务端建立连接，那么管理所有的连接，验证客户端的身份和权限，这些功能就在连接层完成。

**服务层**

连接层会把 SQL 语句交给服务层，这里面又包含一系列的流程：比如**查询缓存**的判断、根据 SQL 调用相应的接口，对我们的 SQL 语句进行**词法和语法的解析**（比如关键字怎么识别，别名怎么识别，语法有没有错误等等）。

然后就是**优化器**，MySQL 底层会根据一定的规则对我们的 SQL 语句进行优化，最

后再交给执行器去执行。

**存储引擎**

存储引擎就是我们的数据真正存放的地方，在 MySQL 里面支持不同的存储引擎。再往下就是内存或者磁盘。

### 更新 SQL 的执行

在数据库里面，我们说的 update 操作其实包括了更新、插入和删除。

更新流程和查询流程有什么不同呢？

基本流程也是一致的，也就是说，它也要经过解析器、优化器的处理，最后交给执行器。区别就在于拿到符合条件的数据之后的操作。

#### 缓冲池 Buffer Pool

首先，InnnoDB 的数据都是放在磁盘上的，InnoDB 操作数据有一个最小的逻辑单位，叫做页（索引页和数据页）。我们对于数据的操作，不是每次都直接操作磁盘，因为磁盘的速度太慢了。

InnoDB 使用了一种缓冲池的技术，也就是把磁盘读到的页放到一块内存区域里面。这个内存区域就叫 Buffer Pool；

下一次读取相同的页，先判断是不是在缓冲池里面，如果是，就直接读取，不用再次访问磁盘。修改数据的时候，先修改缓冲池里面的页。

内存的数据页和磁盘数据不一致的时候，我们把它叫做脏页。InnoDB 里面有专门的后台线程把 Buffer Pool 的数据写入到磁盘，每隔一段时间就一次性地把多个修改写入磁盘，这个动作就叫做刷脏。

##### InnoDB的内存结构和磁盘结构

**内存结构**

主要分为 3 个部分： Buffer Pool、Change Buffer、Adaptive Hash Index，另外还有一个（redo）log buffer。

**Buffer Pool**

缓存页面的信息，包括数据页，索引页；从磁盘加载到内存的最小单位；Buffer Pool 默认大小是 128M（134217728 字节），可以调整。

查看参数（系统变量）：

SHOW VARIABLES like '%innodb\_buffer\_pool%';

脏页：内存中的数据和磁盘中的数据不一致的数据页

刷脏：将内存中的数据和磁盘的数据同步的过程

当数据加载到内存中，如果数据库机器宕机了怎么处理

在InnoDB中设计了一个日志文件redo.log，如果进入内存就将数据记录到redo.log中，当机器宕机后重启就读取redolog来恢复数据(redolog用来做崩溃恢复)

Undolog:撤销日志，记录数据修改之前的状态，用来做事务回滚， Service层的日志 binlog：能够被所有的存储引擎使用，记录所有的操作ddl,dml（记录sql）;实现主从复制,

查看服务器状态，里面有很多跟 Buffer Pool 相关的信息：

**内存的缓冲池写满了怎么办？（Redis 设置的内存满了怎么办？）**

InnoDB 用 LRU算法来管理缓冲池（链表实现，不是传统的 LRU，分成了 young 和old），经过淘汰的数据就是热点数据内存缓冲区对于提升读写性能有很大的作用。

思考一个问题：

当需要更新一个数据页时，如果数据页在 Buffer Pool 中存在，那么就直接更新好了。否则的话就需要从磁盘加载到内存，再对内存的数据页进行操作。也就是说，如果没有命中缓冲池，至少要产生一次磁盘 IO，有没有优化的方式呢？

**Change buffer 写缓冲**

在MySQL5.5之前，叫插入缓冲(insert buffer)，只针对insert做了优化；现在对delete和update也有效，叫做写缓冲(change buffer)。

如果这个数据页不是唯一索引，不存在数据重复的情况，也就不需要从磁盘加载索引页判断数据是不是重复（唯一性检查）。这种情况下可以先把修改记录在内存的缓冲池中，从而提升更新语句（Insert、Delete、Update）的执行速度，最后把 Change Buffer 记录到数据页的操作叫做 merge。

写缓冲的目的是降低写操作的磁盘IO，提升数据库性能。

把 Change Buffer 记录到数据页的操作叫做 merge。什么时候发生 merge？

有几种情况：在访问这个数据页的时候，或者通过后台线程、或者数据库 shut down、redo log 写满时触发。

如果数据库大部分索引都是非唯一索引，并且业务是写多读少，不会在写数据后立刻读取，就可以使用 Change Buffer（写缓冲）。写多读少的业务，调大这个值：

SHOW VARIABLES LIKE 'innodb\_change\_buffer\_max\_size';代表 Change Buffer 占 Buffer Pool 的比例，默认 25%。

**Adaptive Hash Index**

存放hash索引的内存空间

**（redo）Log Buffer**

如果 Buffer Pool 里面的脏页还没有刷入磁盘时，数据库宕机或者重启，这些数据丢失。如果写操作写到一半，甚至可能会破坏数据文件导致数据库不可用。

为了避免这个问题，InnoDB 把所有对页面的修改操作专门写入一个日志文件，并且在数据库启动时从这个文件进行恢复操作（实现 crash-safe）——用它来实现事务的持久性。这个文件就是磁盘的 redo log（叫做重做日志），对应于/var/lib/mysql/目录下的ib\_logfile0 和 ib\_logfile1，每个 48M。

这 种 日 志 和 磁 盘 配 合 的 整 个 过 程 ， 其 实 就 是 MySQL 里 的 WAL 技 术（Write-Ahead Logging），它的关键点就是先写日志，再写磁盘。

写磁盘，为什么不直接写到 db file 里面去？为什么先写日志再写磁盘？

磁盘的最小组成单元是扇区，通常是 512 个字节。

操作系统和内存打交道，最小的单位是页 Page。

操作系统和磁盘打交道，读写磁盘，最小的单位是块 Block

如果我们所需要的数据是随机分散在不同页的不同扇区中，那么找到相应的数据需要等到磁臂旋转到指定的页，然后盘片寻找到对应的扇区，才能找到我们所需要的一块数据，一次进行此过程直到找完所有数据，这个就是**随机 IO**，读取数据速度较慢。

假设我们已经找到了第一块数据，并且其他所需的数据就在这一块数据后边，那么就不需要重新寻址，可以依次拿到我们所需的数据，这个就叫顺序 IO。

刷盘是随机 I/O，而记录日志是顺序 I/O，顺序 I/O 效率更高。因此先把修改写入日志，可以延迟刷盘时机，进而提升系统吞吐。

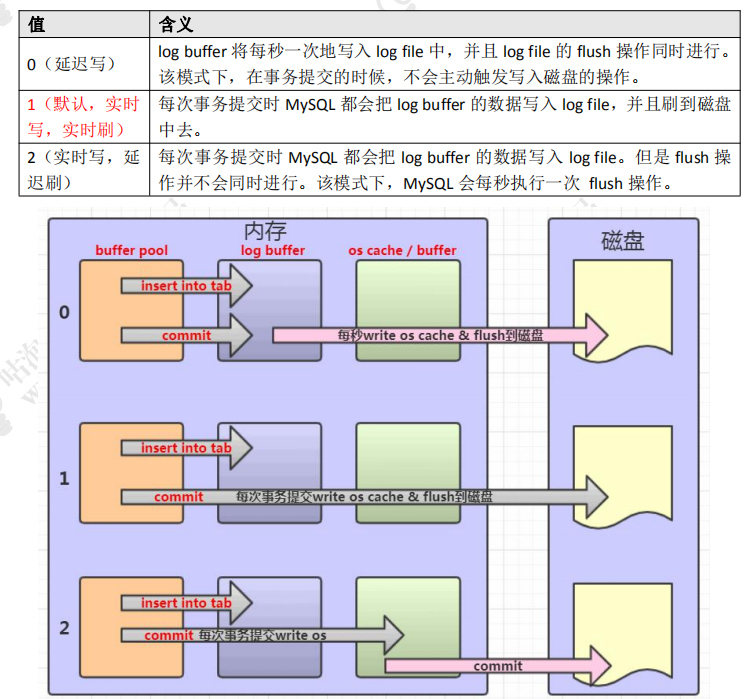
当然 redo log 也不是每一次都直接写入磁盘，在 Buffer Pool 里面有一块内存区域（Log Buffer）专门用来保存即将要写入日志文件的数据，默认 16M，它一样可以节省磁盘 IO。

redo log 的内容主要是用于崩溃恢复。磁盘的数据文件，数据来自 bufferpool。redo log 写入磁盘，不是写入数据文件。

那么，Log Buffer 什么时候写入 log file？

在我们写入数据到磁盘的时候，操作系统本身是有缓存的。flush 就是把操作系统缓冲区写入到磁盘。 log buffer 写入磁盘的时机，由一个参数控制，默认是 1。

SHOW VARIABLES LIKE 'innodb\_flush\_log\_at\_trx\_commit';

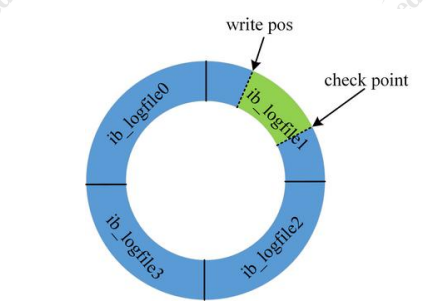


Redo log的特点

1、redo log 是 InnoDB 存储引擎实现的，并不是所有存储引擎都有。

2、不是记录数据页更新之后的状态，而是记录这个页做了什么改动，属于**物理日志（也就是说保存在磁盘上的）**。

3、redo log 的大小是固定的，前面的内容会被覆盖。



check point 是当前要覆盖的位置。如果 write pos 跟 check point 重叠，说明 redo

log 已经写满，这时候需要同步 redo log 到磁盘中。

这是 MySQL 的内存结构，总结一下，分为：Buffer pool、change buffer、Adaptive Hash Index、 log buffer。磁盘结构里面主要是各种各样的表空间，叫做 Table space。

**磁盘结构**

表空间可以看做是 InnoDB 存储引擎逻辑结构的最高层，所有的数据都存放在表空间中。InnoDB 的表空间分为 5 大类。

**系统表空间 system tablespace**

在默认情况下 InnoDB 存储引擎有一个**共享表空间**（对应文件/var/lib/mysql/ibdata1），也叫系统表空间。

InnoDB 系统表空间包含 InnoDB 数据字典和双写缓冲区，Change Buffer 和 Undo Logs

数据字典：由内部系统表组成，存储表和索引的元数据（定义信息）

双写缓冲（InnoDB 的一大特性）：InnoDB 的页和操作系统的页大小不一致，InnoDB 页大小一般为 16K，操作系统页大小为 4K，InnoDB 的页写入到磁盘时，一个页需要分 4 次写;

如果存储引擎正在写入页的数据到磁盘时发生了宕机，可能出现页只写了一部分的情况，比如只写了 4K，就宕机了，这种情况叫做部分写失效（partial page write），可能会导致数据丢失

我们不是有 redo log 吗？但是有个问题，如果这个页本身已经损坏了，用它来做崩溃恢复是没有意义的。所以在对于应用 redo log 之前，需要一个页的副本。如果出现了写入失效，就用页的副本来还原这个页，然后再应用 redo log。这个页的副本就是 double write，InnoDB 的双写技术。通过它实现了数据页的可靠性。跟 redo log 一样，double write 由两部分组成，一部分是内存的 double write，

一个部分是磁盘上的 double write。因为 double write 是顺序写入的，不会带来很大的开销。

在默认情况下，所有的表共享一个系统表空间，这个文件会越来越大，而且它的空间不会收缩。

**独占表空间 file-per-table tablespaces**

让每张表独占一个表空间。这个开关通过 innodb\_file\_per\_table 设置，默认开启

SHOW VARIABLES LIKE 'innodb\_file\_per\_table';

开启后，则每张表会开辟一个表空间，这个文件就是数据目录下的 ibd 文件（例如/var/lib/mysql/gupao/user\_innodb.ibd），存放表的索引和数据。但是其他类的数据，如回滚（undo）信息，插入缓冲索引页、系统事务信息，二次写缓冲（Double write buffer）等还是存放在原来的共享表空间内。

**通用表空间 general tablespaces**

通用表空间也是一种共享的表空间，跟 ibdata1 类似。

可以创建一个通用的表空间，用来存储不同数据库的表，数据路径和文件可以自定

义。语法：

create tablespace ts2673 add datafile '/var/lib/mysql/ts2673.ibd' file\_block\_size=16K engine=innodb;

在创建表的时候可以指定表空间，用 ALTER 修改表空间可以转移表空间。

create table t2673(id integer) tablespace ts2673;

不同表空间的数据是可以移动的。

删除表空间需要先删除里面的所有表：

drop table t2673;

drop tablespace ts2673;

**临时表空间 temporary tablespaces**

存储临时表的数据，包括用户创建的临时表，和磁盘的内部临时表。对应数据目录下的 ibtmp1 文件。当数据服务器正常关闭时，该表空间被删除，下次重新产生。

**undo log tablespace**

undo log（撤销日志或回滚日志）记录了事务发生之前的数据状态（不包括 select）。如果修改数据时出现异常，可以用 undo log 来实现回滚操作（保持原子性）。

在执行 undo 的时候，仅仅是将数据从逻辑上恢复至事务之前的状态，而不是从物理页面上操作实现的，属于逻辑格式的日志。

redo Log 和 undo Log 与事务密切相关，统称为事务日志。

undo Log 的数据默认在系统表空间 ibdata1 文件中，因为共享表空间不会自动收缩，也可以单独创建一个 undo 表空间。

show global variables like '%undo%';

更新操作的流程，这是一个简化的过程。

name 原值是 lei1。

update user set name = 'penyuyan' where id=1;

1、事务开始，从内存或磁盘取到这条数据，返回给 Server 的执行器；

2、执行器修改这一行数据的值为 penyuyan；

3、记录 name=qingshan 到 undo log；

4、记录 name=penyuyan 到 redo log；

5、调用存储引擎接口，在内存（Buffer Pool）中修改 name=penyuyan；

6、事务提交。

**后台线程**

后台线程的主要作用是负责刷新内存池中的数据和把修改的数据页刷新到磁盘。

后台线程分为：master thread，IO thread，purge thread，page cleaner thread。

master thread 负责刷新缓存数据到磁盘并协调调度其它后台进程。

IO thread 分为 insert buffer、log、read、write 进程。分别用来处理 insert buffer、

重做日志、读写请求的 IO 回调。purge thread 用来回收 undo 页。

page cleaner thread 用来刷新脏页。

### Binlog

除了 InnoDB 架构中的日志文件，MySQL 的 Server 层也有一个日志文件，叫做binlog，它可以被所有的存储引擎使用。

binlog 以事件的形式记录了所有的 DDL 和 DML 语句（因为它记录的是操作而不是数据值，属于逻辑日志），可以用来做主从复制和数据恢复。跟 redo log 不一样，它的文件内容是可以追加的，没有固定大小限制。

在开启了 binlog 功能的情况下，我们可以把 binlog 导出成 SQL 语句，把所有的操作重放一遍，来实现数据的恢复。

binlog 的另一个功能就是用来实现主从复制，它的原理就是从服务器读取主服务器的 binlog，然后执行一遍。

有了这两个日志之后，我们来看一下一条更新语句是怎么执行的：

例如一条语句：update teacher set name='盆鱼宴' where id=1;

1、先查询到这条数据，如果有缓存，也会用到缓存。

2、把 name 改成盆鱼宴，然后调用引擎的 API 接口，写入这一行数据到内存，同时记录 redo log。这时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，可以随时提交。

3、执行器收到通知后记录 binlog，然后调用存储引擎接口，设置 redo log为 commit状态。

4、更新完成。

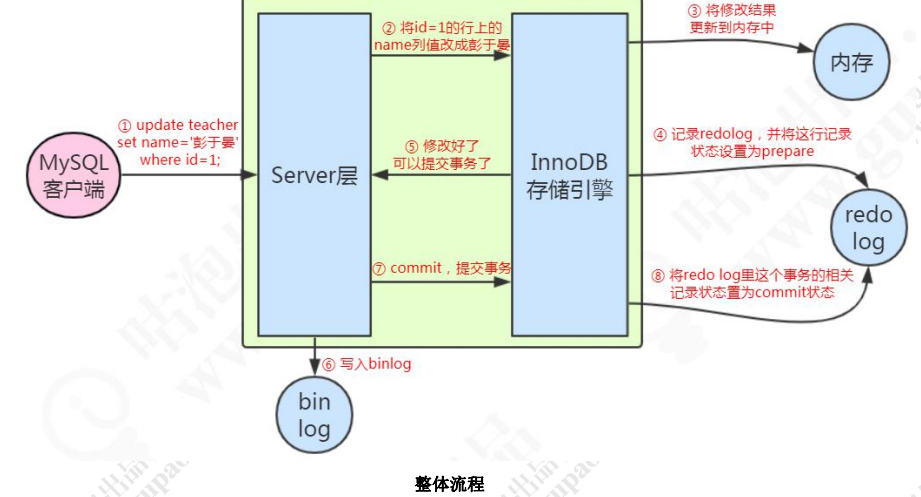
这张图片的重点（没必要背下来）：

1、先记录到内存，再写日志文件

2、记录 redo log 分为两个阶段。

3、存储引擎和 Server 记录不同的日志。

3、先记录 redo，再记录 binlog。



# MySql索引

## 索引是什么

### 定义

数据库索引，是数据库管理系统（DBMS）中一个排序的数据结构，以协助快速查询数据库表中数据。

首先数据是以文件的形式存放在磁盘上面的，每一行数据都有它的磁盘地址。如果没有索引的话，要从 500 万行数据里面检索一条数据，只能依次遍历这张表的全部数据，直到找到这条数据。

但是有了索引之后，只需要在索引里面去检索这条数据就行了，因为它是一种特殊的专门用来快速检索的数据结构，我们找到数据存放的磁盘地址以后，就可以拿到数据了。

就像我们从一本 500 页的书里面去找特定的一小节的内容，肯定不可能从第一页开始翻。那么这本书有专门的目录，它可能只有几页的内容，它是按页码来组织的，可以根据拼音或者偏旁部首来查找，只要确定内容对应的页码，就能很快地找到我们想要的内容。

### 索引类型

在 InnoDB 里面，索引类型有三种，普通索引、唯一索引（主键索引是特殊的唯一索引）、全文索引。

**普通（Normal）**：也叫非唯一索引，是最普通的索引，没有任何的限制。

**唯一（Unique）**：唯一索引要求键值不能重复。另外需要注意的是，主键索引是一种特殊的唯一索引，它还多了一个限制条件，要求键值不能为空。主键索引用 primay key创建。

主键索引是唯一索引的一种，主键索引要求值不能重复，并且不能为空；

**全文（Fulltext）**：针对比较大的数据，比如我们存放的是消息内容，有几 KB 的数据的这种情况，如果要解决 like 查询效率低的问题，可以创建全文索引。只有文本类型的字段才可以创建全文索引，比如 char、varchar、text。

create table m3 (

name varchar(50),

fulltext index(name)

);

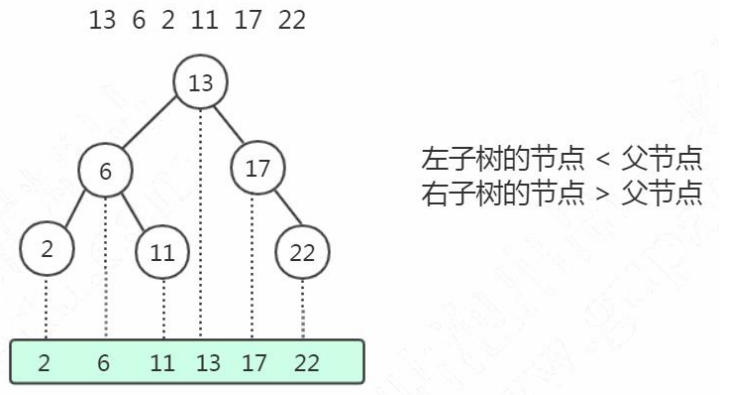
## 索引的结构

### 二分查找

二分查找的思想就是折半查找，每次把查找的数据缩小一半，如果数据已经排好了顺序，那么查找的效率就非常高

有序数组的等值查询和比较查询效率非常高，但是更新数据的时候会出现一个问题，可能要挪动大量的数据（改变 index），所以只适合存储静态的数据。

### 二叉树查找



特点：左子树所有的节点都小于父节点，右子树所有的节点都大于父节点。投影到平面以后，就是一个有序的线性表

二叉查找树既能够实现快速查找，又能够实现快速插入。

但是二叉查找树有一个问题：就是它的查找耗时是和这棵树的深度相关的，在最坏的情况下时间复杂度会退化成O(n)。

什么情况是最坏的情况呢？

当插入的数字的顺序刚好是有序的，这个时候二叉树就会变成链表（我们把这种树叫做“斜树”），这种情况下不能达到加快检索速度的目的，和顺序查找效率是没有区别的；

左右子树深度差太大，这棵树的左子树根本没有节点——也就是它不够平衡



### 平衡二叉树查找(Avl树)

平衡二叉树的定义：左右子树深度差绝对值不能超过 1。

是什么意思呢？比如左子树的深度是 2，右子树的深度只能是 1 或者 3。

平衡二叉树也不适合做索引的数据结构

因为它只有2个节点，用它来作为索引的数据结构，那么树的深度将会非常大，那遍历索引的时候，将会进行很多次的磁盘io

当我们用树的结构来存储索引的时候，访问一个节点就要跟磁盘之间发生一次 IO。

InnoDB 操作磁盘的最小的单位是一页（或者叫一个磁盘块），大小是 16K(16384 字节)。

那么，一个树的节点就是 16K 的大小。

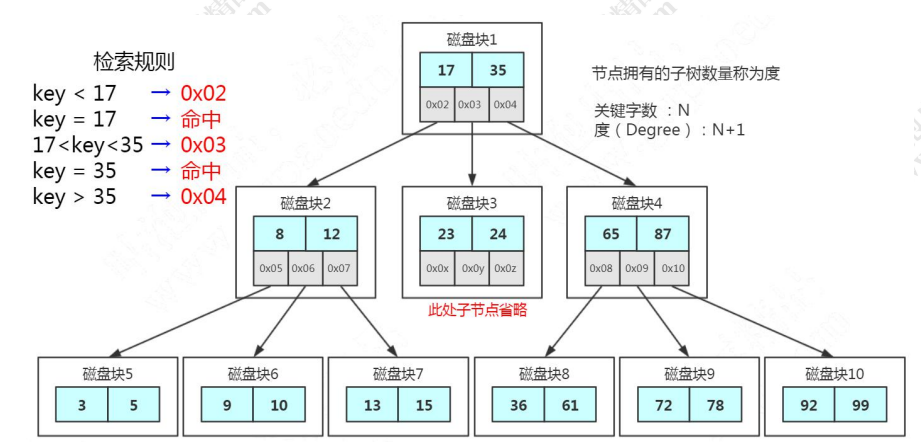
如果我们一个节点只存一个键值+数据+引用，例如整形的字段，可能只用了十几个

或者几十个字节，它远远达不到 16K 的容量，所以访问一个树节点，进行一次 IO 的时候，浪费了大量的空间。

所以如果每个节点存储的数据太少，从索引中找到我们需要的数据，就要访问更多的节点，意味着跟磁盘交互次数就会过多。

### B tree

多路平衡查找树，叫做 B Tree（B 代表平衡）。B 树在枝节点和叶子节点存储键值、数据地址、节点引用。它有一个特点：分叉数（路数,指向子节点的指针）永远比关键字数多 1。比如我们画的这棵树，每个节点存储两个关键字，那么就会有三个指针指向三个子节点。

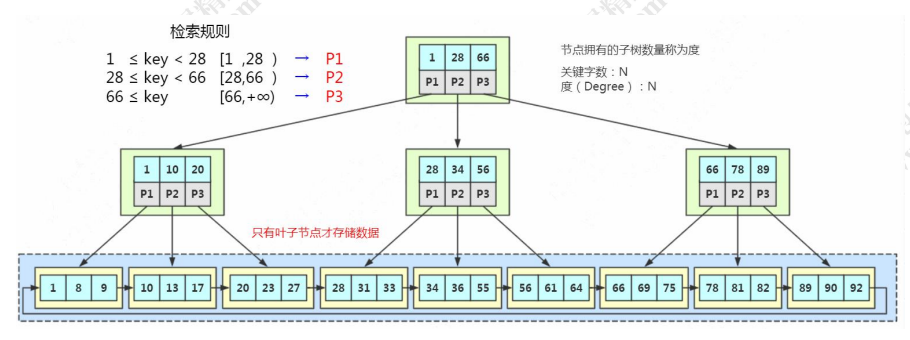


### B+Tree

B Tree 的效率已经很高了，为什么 MySQL 还要对 B Tree 进行改良，最终使用了B+Tree 呢？

总体上来说，这个 B 树的改良版本解决的问题比 B Tree 更全面。

InnoDB 里面的 B+树的存储结构：



**MySQL 中的 B+Tree 有几个特点：**

**1、它的关键字的数量是跟路数相等的；**

**2、B+Tree 的根节点和枝节点中都不会存储数据，只有叶子节点才存储数据。**搜索到关键字不会直接返回，会到最后一层的叶子节点。比如我们搜索 id=28，虽然在第一层直接命中了，但是全部的数据在叶子节点上面，所以我还要继续往下搜索，一直到叶子节点。

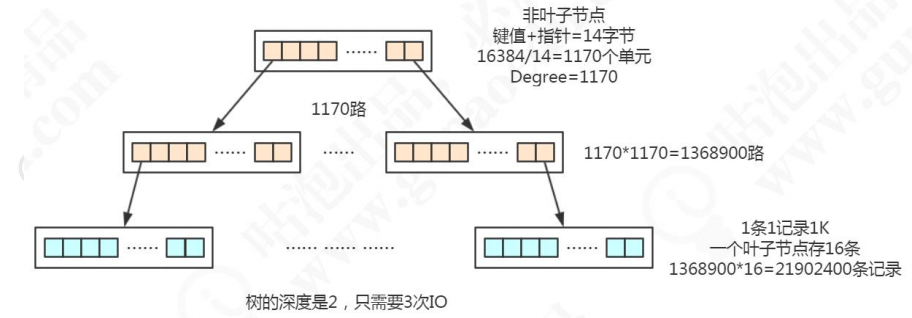
举个例子：假设一条记录是 1K，一个叶子节点（一页）可以存储 16 条记录。非叶

子节点可以存储多少个指针？

1kb=1024byte

1M=1024kb

假设索引字段是 bigint 类型，长度为 8 字节。指针大小在 InnoDB 源码中设置为6 字节，这样一共 14 字节。非叶子节点（一页）可以存储 16384/14=1170 个这样的单元（键值+指针），代表有 1170 个指针。树 深 度 为 2 的 时 候 ， 有 1170^2 个 叶 子 节 点 ， 可 以 存 储 的 数 据 为1170\*1170\*16=21902400



在查找数据时一次页的查找代表一次 IO，也就是说，一张 2000 万左右的表，查询

数据最多需要访问 3 次磁盘。

所以在 InnoDB 中 B+ 树深度一般为 1-3 层，它就能满足千万级的数据存储。

**3、B+Tree 的每个叶子节点增加了一个指向相邻叶子节点的指针，它的最后一个数**

**据会指向下一个叶子节点的第一个数据，形成了一个有序链表的结构。排序能力更强（因为叶子节点上有下一个数据区的指针，数据形成了链表）**

**4、扫库、扫表能力更强（如果我们要对表进行全表扫描，只需要遍历叶子节点就可以了，不需要遍历整棵 B+Tree 拿到所有的数据）**

**5、效率更加稳定（B+Tree 永远是在叶子节点拿到数据，所以 IO 次数是稳定的**

### InnoDB逻辑存储结构

MySQL 的存储结构分为 5 级：表空间、段、簇、页、行。

**表空间 Table Space**：表空间可以看做是 InnoDB 存储引擎逻辑结构的

最高层，所有的数据都存放在表空间中。分为：系统表空间、独占表空间、通用表空间、

临时表空间、Undo 表空间。

**段 Segment**

表空间是由各个段组成的，常见的段有数据段、索引段、回滚段等，段是一个逻辑的概念。一个 ibd 文件（独立表空间文件）里面会由很多个段组成。

创建一个索引会创建两个段，一个是索引段：leaf node segment，一个是数据段：

non-leaf node segment。索引段管理非叶子节点的数据。数据段管理叶子节点的数据。

也就是说，一个表的段数，就是索引的个数乘以 2。

**簇**

一个段（Segment）又由很多的簇（也可以叫区）组成，每个区的大小是 1MB（64个连续的页）。每一个段至少会有一个簇，一个段所管理的空间大小是无限的，可以一直扩展下去，但是扩展的最小单位就是簇

**页**

为了高效管理物理空间，对簇进一步细分，就得到了页。簇是由连续的页（Page）组成的空间，一个簇中有 64 个连续的页。 （1MB／16KB=64）。这些页面在物理上和逻辑上都是连续的。InnoDB 中也有页（也可以称为块），每个页默认 16KB。页是 InnoDB 存储引擎磁盘管理的最小单位，通过 innodb\_page\_size 设置。

一个表空间最多拥有 2^32 个页，默认情况下一个页的大小为 16KB，也就是说一个表空间最多存储 64TB 的数据。

注意，文件系统中，也有页的概念。操作系统和内存打交道，最小的单位是页 Page。文件系统的内存页通常是 4K。

SHOW VARIABLES LIKE 'innodb\_page\_size';

假设一行数据大小是 1K，那么一个数据页可以放 16 行这样的数据。

往表中插入数据时，如果一个页面已经写完，产生一个新的叶页面。如果一个簇的所有的页面都被用完，会从当前页面所在段新分配一个簇。如果数据不是连续的，往已经写满的页中插入数据，会导致叶页面分裂：

**行**

InnoDB 存储引擎是面向行的（row-oriented），也就是说数据的存放按行进行存放。

### 红黑树

为什么不用红黑树？1、只有两路；2、不够平衡。

红黑树一般只放在内存里面用。例如 Java 的 TreeMap

## 索引的实现

### Mysql数据存储文件

InnoDB：存储引擎为InnoDB的表有两个文件.frm和.ibd

Myisam：存储引擎为myisam的表有三个数据文件.frm和myd,myi

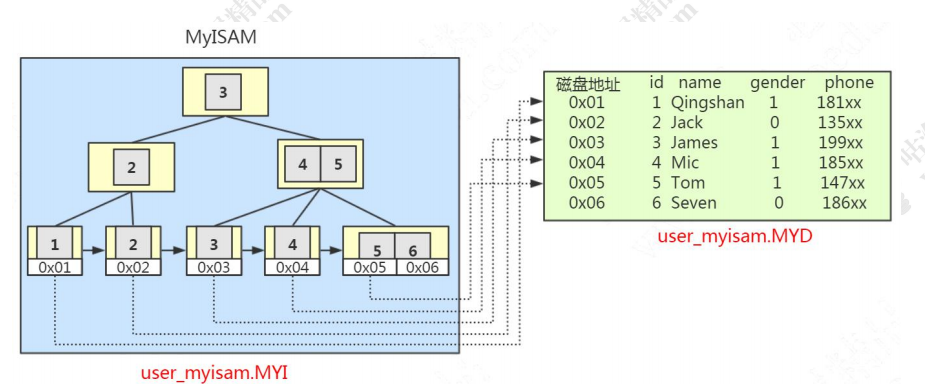
.frm 是 MySQL 里面表结构定义的文件，不管你建表的时候选用任何一个存储引擎都会生成。

在 MyISAM 里面，另外有两个文件：一个是.MYD 文件，D 代表 Data，是 MyISAM 的数据文件，存放数据记录。一个是.MYI 文件，I 代表 Index，是 MyISAM 的索引文件，存放索引，比如我们在id 字段上面创建了一个主键索引，那么主键索引就是在这个索引文件里面。也就是说，在 MyISAM 里面，索引和数据是两个独立的文件。

在InnoDB里面索引和数据都是放在ibd文件中的

### Myisam

MyISAM 的 B+Tree 里面，叶子节点存储的是数据文件对应的磁盘地址。所以从索引文件.MYI 中找到键值后，会到数据文件.MYD 中获取相应的数据记录



这里是主键索引，如果是辅助索引，有什么不一样呢？

在 MyISAM 里面，辅助索引也在这个.MYI 文件里面。辅助索引跟主键索引存储和检索数据的方式是没有任何区别的，一样是在索引文件里面找到磁盘地址，然后到数据文件里面获取数据

### InnoDB

在 InnoDB 里面，它是以主键为索引来组织数据的存储的，所以索引文件和数据文件是同一个文件，都在.ibd 文件里面。在 InnoDB 的主键索引的叶子节点上，它直接存储了我们的数据。

**以主键为索引来组织数据的存储**

**聚集索引**：就是索引键值的逻辑顺序和表数据行的物理存储顺序是一样的（比如字典的目录是按拼音排序的，内容也是按拼音排序的，按拼音排序的这种目录就叫聚集索引）

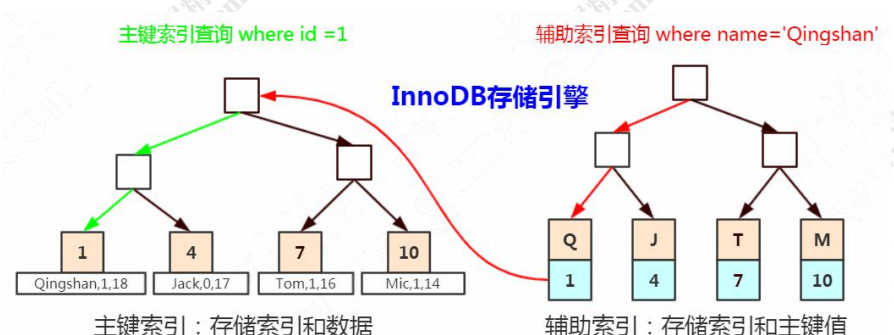
在 InnoDB 里面，它组织数据的方式叫做（聚集）索引组织表（clustered index organize table），所以主键索引是聚集索引，非主键都是非聚集索引。

一个表只能有一个聚集索引，因为一个表的物理顺序只有一种情况，所以，对应的聚集索引只能有一个

**非聚集索引**

定义：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。如果某索引不是聚集索引，则表中的行物理顺序与索引顺序不匹配，与非聚集索引相比，聚集索引有着更快的检索速度。

普通索引怎么检索数据



InnoDB 中，主键索引和辅助索引是有一个主次之分的。

辅助索引存储的是辅助索引和主键值。如果使用辅助索引查询，会根据主键值在主键索引中查询，最终取得数据。

如果一张表没有主键怎么办？

1、如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引。

2、如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引作为主键索引。

3、如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐藏的聚集索引，它会随着行记录的写入而主键递增。

### Hash索引

## 索引的使用原则

### 列的离散度

列的全部不同值和所有数据行的比例；简单来说，如果列的重复值越多，离散度就越低，重复值越少，离散度就越高。

当我们在重复值多的列上建立的索引去检索数据的时候，由于重复值太多，需要扫描的行数就更多。如果在 B+Tree 里面的重复值太多，MySQL 的优化器发现走索引跟使用全表扫描差不了多少的时候，就算建了索引，也不一定会走索引。

所以建立索引，要使用离散度（选择度）更高的字段。

### 联合索引最左匹配

在多条件查询的时候，会创建联合索引。单列索引可以看成是特殊的联合索引。联合索引在 B+Tree 中是复合的数据结构，它是按照从左到右的顺序来建立搜索树的，在建立联合索引的时候，一定要把最常用的列放在最左边

如果我们创建三个字段的索引 index(a,b,c)，相当于创建三个索引：

index(a)

index(a,b)

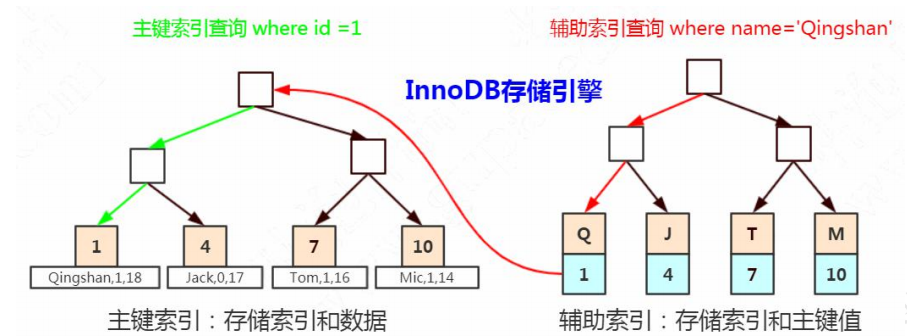
index(a,b,c)

用 where b=? 和 where b=? and c=? 和 where a=? and c=?是不能使用到索引的。不能不用第一个字段，不能中断。这里就是 MySQL 联合索引的最左匹配原则

### 覆盖索引

回表：

非主键索引，我们先通过索引找到主键索引的键值，再通过主键值查出索引里面没有的数据，它比基于主键索引的查询多扫描了一棵索引树，这个过程就叫回表。



在辅助索引里面，不管是单列索引还是联合索引，如果 select 的数据列只用从索引中就能够取得（查询的字段是索引），不必从数据区中读取，这时候使用的索引就叫做覆盖索引，这样就避免了回表。

因为覆盖索引减少了 IO 次数，减少了数据的访问量，可以大大地提升查询效率

### 前缀索引

当字段值比较长的时候，建立索引会消耗很多的空间，搜索起来也会很慢。我们可以通过截取字段的前面一部分内容建立索引，这个就叫前缀索引。

问题是，截取多少呢？截取得多了，达不到节省索引存储空间的目的，截取得少了，

重复内容太多，字段的散列度（选择性）会降低。怎么计算不同的长度的选择性呢？

先看一下字段在全部数据中的选择度：

### 索引的使用

创建

1. 在用于where判断order排序和join的(on)字段创建索引
2. 索引的个数不要过多—浪费空间，更新变慢
3. 区分度低的字段不要建字段—离散度低的，导致扫描行数变多
4. 频繁更新的值，不要作为主键或者索引---页分裂
5. 组合索引把列离散度高（区分度）的放在最前面
6. 创建复核索引，而不是修改单列索引
7. 过长的字段，创建全文索引，前缀索引，hash索引
8. 不建议用无序的值（uuid）作为索引--无序会产生页分裂、递增的ID离散度较高，不会重复

索引无效

1. 索引列上使用函数（replace\SUBSTR\CONCAT\sum count avg）、表达式、计算（+ - \* /）
2. 字符串不加引号，出现隐式转换
3. like 条件中前面带%：过滤的开销太大，所以无法使用索引。这个时候可以用全文索引
4. 负向查询：NOT LIKE 不能：

# Mysql的事务

## 事务定义

事务是数据库管理系统（DBMS）执行过程中的一个逻辑单位，由一系列的数据库操作构成的。

这里面有两个关键点，第一个，它是数据库最小的工作单元，是不可以再分的。第二个，它可能包含了一个或者一系列的 DML 语句，包括 insert delete update。

（单条 DDL（create drop）和 DCL（grant revoke）也会有事务）

Mysql中只有InnoDB支持事务，一个也是InnoDB称为mysql默认存储引擎的主要原因

## 事务的4大特性

1. 原子性:对数据库的一系列的操作，要么都成功，要么都失败，不可能出现部分成功或者部分失败的情况

事务的操作全部失败：事务回滚。

在 InnoDB 里面是通过 undo log 来实现的，它记录了数据修改之前的值（逻辑日志），一旦发生异常，就可以用 undo log 来实现回滚操作。

1. 一致性：指的是数据库的完整性约束没有被破坏，**事务执行的前后都是合法的数据状态**。比如主键必须是唯一的，字段长度符合要求。除了数据库自身的完整性约束，还有一个是用户自定义的完整性。

比如说转账的这个场景，A 账户余额减少 1000，B 账户余额只增加了 500，这个时候因为两个操作都成功了，按照我们对原子性的定义，它是满足原子性的， 但是它没有满足一致性，因为它导致了会计科目的不平衡。

1. 隔离性：有了事务的定义以后，在数据库里面会有很多的事务同时去操作同一张表或者同一行数据，必然会产生一些并发或者干扰的操作；

**隔离性的定义，就是这些很多个的事务，对表或者行的并发操作，应该是透明的，互相不干扰的**

1. 持久性：对数据库的任意的操作，增删改，只要事务提交成功，那么结果就是永久性的，不可能因为我们系统宕机或者重启了数据库的服务器，它又恢复到原来的状态了

持久性的实现：

持久性是通过 redo log 和 double write 双写缓冲来实现的，我们操作数据的时候，会先写到内存的 buffer pool 里面，同时记录 redo log，如果在刷盘之前出现异常，在重启后就可以读取 redo log 的内容，写入到磁盘，保证数据的持久性。当然，恢复成功的前提是数据页本身没有被破坏，是完整的，这个通过双写缓冲（double write）保证。

## 数据库开启事务

InnoDB 里面有一个 autocommit 的参数（分成两个级别， session 级别和 global

级别）。

show variables like 'autocommit';

它的默认值是 ON。

如果它的值是 true/on 的话，我们在操作数据的时候，会自动开启一个事务，和自动提交事务。

如果我们把 autocommit 设置成 false/off，那么数据库的事务就需要我们手动地去开启和手动地去结束。

### 自动开启事务

Autocommit：参数为on时，自动开启和自动提交

这样执行一条更新语句的时候，它有事务

### 手动开启事务

一种是用 begin；

一种是用 start transaction。

手动结束事务

1. 提交事务：commit命令
2. 回滚事务：rollback回滚

当客户端断开连接的时候事务也会结束

### 事务并发

事务并发带来的问题

1. 脏读

在一个事务里面，由于其他的事务修改了数据并且没有提交，而导致了前后两次读取数据不一致的情况，也就是读取到其他事务未提交的数据

1. 不可重复读

在两个事务中，第一个事务读取到一条数据，另一个事务中修改了这条数据，并且提交了事务，此时第一个事务再次读取到这条数据，这样前后两次读取数据一样的情况就叫做不可重复读

1. 幻读

在第一个事务里面我们执行了一个范围查询，这个时候满足条件的数据只有一条。在第二个事务里面，它插入了一行数据，并且提交了。重点：插入了一行数据。在第一个事务里面再去查询的时候，它发现多了一行数据，这种情况就是叫做幻读

不可重复读和幻读的区别

不可重复读是修改或者删除，幻读是插入。

### 隔离级别

无论是脏读，还是不可重复读，还是幻读，它们都是数据库的读一致性的问题，都是在一个事务里面前后两次读取出现了不一致的情况。

读一致性的问题，必须要由数据库提供一定的事务隔离机制来解决

#### 4 个隔离级别

1. 未提交读（read uncommitted）:一个事务可以读取到其他事务未提交的数据，会出现脏读，所以叫做 RU，它没有解决任何的问题。
2. 已提交读（read committed）:一个事务只能读取到其他事务已提交的数据，不能读取到其他事务未提交的数据，它解决了脏读的问题，但是会出现不可重复读的问题。
3. 可重复读（repeatable read）:它解决了不可重复读的问题，也就是在同一个事务里面多次读取同样的数据结果是一样的，但是在这个级别下，没有定义解决幻读的问题。
4. 串行化（serializable）:所有的事务都是串行执行的，也就是对数据的操作需要排队，已经不存在事务的并发操作了，所以它解决了所有的问题。

#### Mysql InnoDB对隔离级别的支持



InnoDB 支持的四个隔离级别，隔离级别越高，事务的并发度就越低。唯一的区别就在于，InnoDB 在 RR 的级别就**解决了幻读**的问题。这个也是**InnoDB 默认使用 RR 作为事务隔离级别的原因**，既保证了数据的一致性，又支持较高的并发度。

#### 解决读一致性的问题

1. 基于锁的并发控制(LBCC)

要保证前后两次读取数据一致，那么读取数据的时候，锁定要操作的数据，不允许其他的事务修改就行了.

如果仅仅是基于锁来实现事务隔离，一个事务读取的时候不允许其他时候修改，那就意味着不支持并发的读写操作，而我们的大多数应用都是读多写少的，这样会极大地影响操作数据的效率。

1. 多版本并发控制（MVCC）

如果要让一个事务前后两次读取的数据保持一致，那么我们可以在修改数据的时候给它建立一个备份或者叫快照，后面再来读取这个快照就行了

MVCC 的核心思想是： 我可以查到在我这个事务开始之前已经存在的数据，即使它在后面被修改或者删除了。在我这个事务之后新增的数据，我是查不到的。

**InnoDB 为每行记录都实现了两个隐藏字段**：

**DB\_TRX\_ID**，6 字节：插入或更新行的最后一个事务的事务 ID，事务编号是自动递增的（我们把它理解为创建版本号，在数据新增或者修改为新数据的时候，记录当前事务 ID）。

**DB\_ROLL\_PTR**，7 字节：回滚指针（我们把它理解为删除版本号，数据被删除或记录为旧数据的时候，记录当前事务 ID）。

我们把这两个事务 ID 理解为版本号。分别用来存放新增修改和删除执行操作所在事务的事务id；查询的时候这两个字段的值不变

MVCC 的查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

第一个事务，初始化数据（检查初始数据）

Transaction 1

begin;

insert into mvcctest values(NULL,'qingshan') ;

insert into mvcctest values(NULL,'jack') ;

commit;

此时的数据，创建版本是当前事务 ID，删除版本为空：



第二个事务，执行第 1 次查询，读取到两条原始数据，这个时候事务 ID 是 2：

Transaction 2

begin;

select \* from mvcctest ; -- (1) 第一次查询

第三个事务，插入数据：

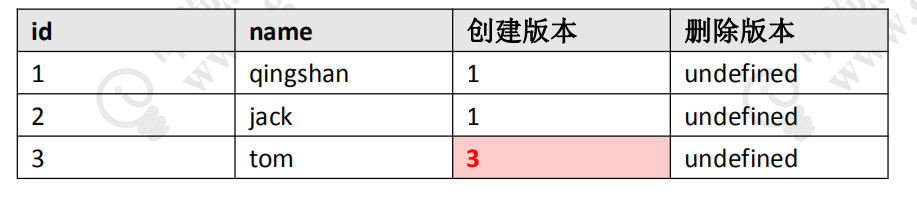
Transaction 3

begin;

insert into mvcctest values(NULL,'tom') ;

commit;

此时的数据，多了一条 tom，它的创建版本号是当前事务编号，3：



第二个事务，执行第 2 次查询：

Transaction 2

select \* from mvcctest ; (2) 第二次查询

MVCC 的查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

也就是不能查到在我的事务开始之后插入的数据，tom 的创建 ID 大于 2，所以还是只能查到两条数据。

第四个事务，删除数据，删除了 id=2 jack 这条记录：

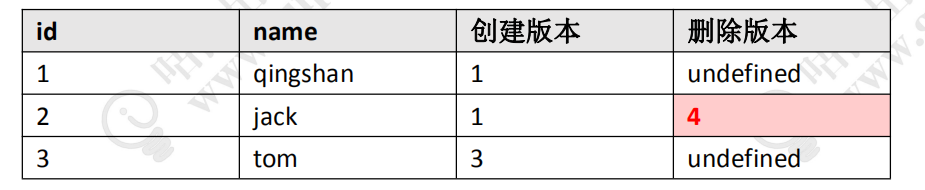
Transaction 4

begin;

delete from mvcctest where id=2;

commit;

此时的数据，jack 的删除版本被记录为当前事务 ID，4，其他数据不变：



在第二个事务中，执行第 3 次查询：

Transaction 2

select \* from mvcctest ; (3) 第三次查询

查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

也就是，在我事务开始之后删除的数据，所以 jack 依然可以查出来。所以还是这两条数据。

第五个事务，执行更新操作，这个事务事务 ID 是 5：

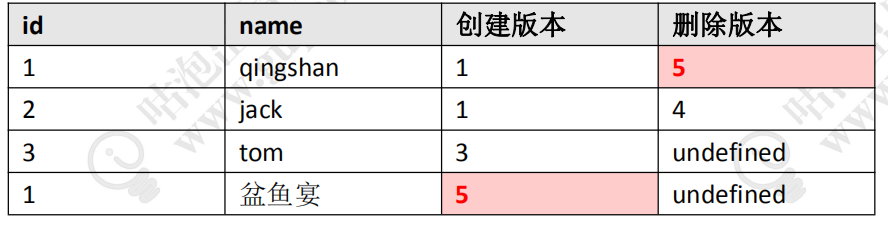
Transaction 4

begin;

update mvcctest set name ='盆鱼宴' where id=1;

commit;

此时的数据，更新数据的时候，旧数据的删除版本被记录为当前事务 ID 5（undo），产生了一条新数据，创建 ID 为当前事务 ID 5：



第二个事务，执行第 4 次查询：

Transaction 2

select \* from mvcctest ;

(4) 第四次查询

查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事

务 ID 的行（或未删除）。

因为更新后的数据 penyuyan 创建版本大于 2，代表是在事务之后增加的，查不出来。而旧数据 qingshan 的删除版本大于 2，代表是在事务之后删除的，可以查出来。

通过以上演示我们能看到，通过版本号的控制，无论其他事务是插入、修改、删除，第一个事务查询到的数据都没有变化。在 InnoDB 中，MVCC 是通过 Undo log 实现的。

# MySql中的锁

在 InnoDB 中，MVCC 和锁是协同使用的，这两种方案并不是互斥的。第一大类解决方案是锁，锁又是怎么实现读一致性的呢？

## MySQL InnoDB 锁的基本类型

### InnoDB的锁类型

Shared and Exclusive Locks: 共享锁和独占锁

Intention Locks:意向锁

Record Locks：记录锁

Gap Locks：间隙锁

Next-Key Locks：临间锁

Insert Intention Locks：插入意向锁

AUTO-INC Locks：AUTO-INC锁

Predicate Locks for Spatial Indexes：空间索引的Predicate Locks（空间索引的谓词锁）

锁的基本模式：

把前面的两个行级别的锁（Shared and ExclusiveLocks），和两个表级别的锁（Intention Locks）称为锁的基本模式。

三个 Record Locks、Gap Locks、Next-Key Locks，把它们叫做锁的算法，

### 锁的粒度

InnoDB 里面既有行级别的锁，又有表级别的锁；

表锁：锁住一张表，

行锁：就是锁住表里面的一行数据；

表锁的粒度肯定大于行锁，表锁的加锁效率高，表锁只需要直接锁住表就可以了，而行锁好需要去检索这一行数据，所有表锁的加锁效率更高

表锁的冲突比行锁大，当我们锁住一张表的时候，其他任何一个事务都不能操作这张表。但是我们锁住了表里面的一行数据的时候，其他的事务还可以来操作表里面的其他没有被锁定的行，所以表锁的冲突概率更大

InnoDB 里面我们知道它既支持表锁又支持行锁，另一个常用的存储引擎 MyISAM 支持什么粒度的锁？这是第一个问题。第二个就是 InnoDB 已经支持行锁了，那么它也可以通过把表里面的每一行都锁住来实现表锁，为什么还要提供表锁呢？

要搞清楚这个问题，我们就要来了解一下 InnoDB 里面的基本的锁的模式（lockmode），这里面有两个行锁和两个表锁。

### 锁的基本模式

共享锁

Shared Locks （共享锁），我们获取了一行数据的读锁以后，可以用来读取数据，所以它也叫做读锁，注意不要在加上了读锁以后去写数据，不然的话可能会出现死锁的情况。而且多个事务可以共享一把读锁

用**select …… lock in share mode**; 的方式手工加上一把读锁

释放锁有两种方式，只要事务结束，锁就会自动事务，包括提交事务和结束事务

排他锁

第二个行级别的锁叫做 **Exclusive Locks**（排它锁），它是用来操作数据的，所以又叫做写锁。

只要一个事务获取了一行数据的排它锁，其他的事务就不能再获取这一行数据的共享锁和排它锁。

排它锁的加锁方式有两种

第一种是**自动加排他锁**。我们在操作数据的时候，包括增删改，都会默认加上一个排它锁。

还有一种是**手工加锁**，我们用一个 **FOR UPDATE** 给一行数据加上一个排它锁

释放锁的方式跟前面是一样的

**共享锁和排他锁是行级别的**

意向锁

是由数据库自己维护的。也就是说，当我们给一行数据加上共享锁之前，数据库会自动在这张表上面加一个意向共享锁。当我们给一行数据加上排他锁之前，数据库会自动在这张表上面加一个意向排他锁。

反过来说：如果一张表上面至少有一个意向共享锁，说明有其他的事务给其中的某些数据行加上了共享锁。

如果一张表上面至少有一个意向排他锁，说明有其他的事务给其中的某些数据行加

上了排他锁

意向锁是表级别的

**意向锁的作用**

第一个，我们有了表级别的锁，在InnoDB 里面就可以支持更多粒度的锁

如果说没有意向锁的话，当我们准备给一张表加上表锁的时候，我们首先要做什么？是不是必须先要去判断有没其他的事务锁定了其中了某些行？如果有的话，肯定不能加上表锁。那么这个时候我们就要去扫描整张表才能确定能不能成功加上一个表锁，如果数据量特别大，比如有上千万的数据的时候，加表锁的效率是不是很低？

但是我们引入了意向锁之后就不一样了。我只要判断这张表上面有没有意向锁，如果有，就直接返回失败。如果没有，就可以加锁成功

第二个**InnoDB 里面的意向锁，我们可以把它理解成一个标志是用来提高加锁的效率的。**

**共享锁，排他锁，意向共享锁，意向排他锁就是 MySQL 里面的 4 种基本的锁的模式，或者叫做锁的类型。**

### 行锁的原理

锁是用来解决事务对数据的并发访问的问题的。那么，锁到底锁住了什么呢

InnoDB 的行锁，就是通过锁住索引来实现的。

1、为什么表里面没有索引的时候，锁住一行数据会导致锁表？或者说，如果锁住的是索引，一张表没有索引怎么办？

当一张表没有索引的时候

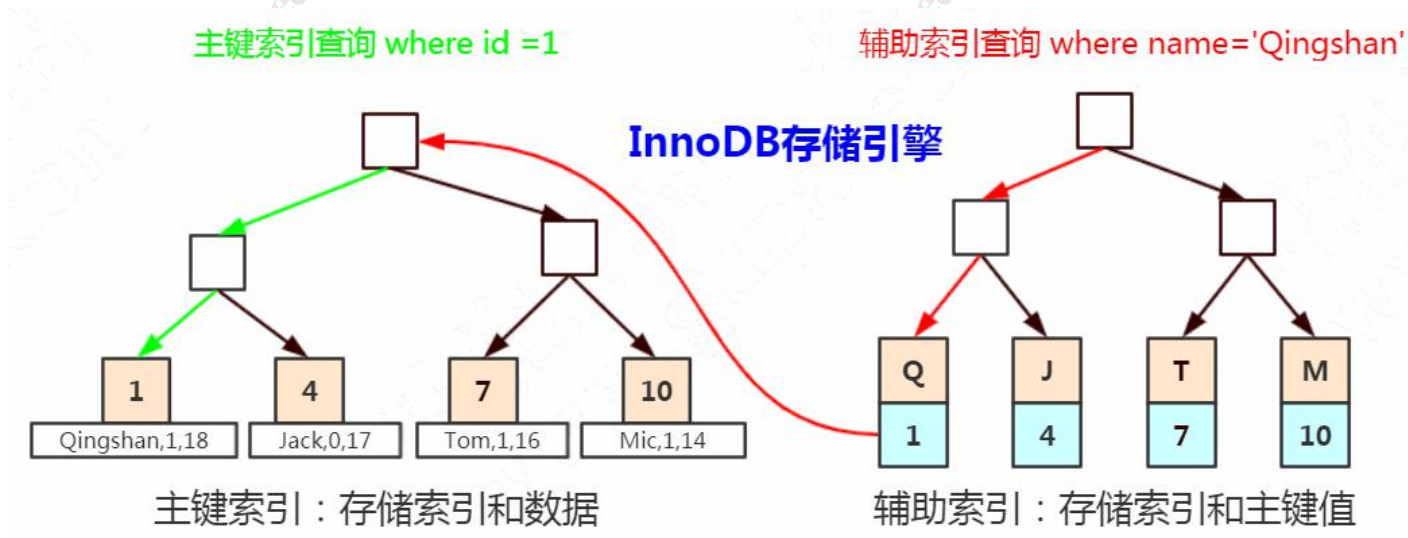
1. 如果定义了主键，InnoDB会选择主键作为聚集索引
2. 如果没有显示的定义主键，InnoDB会选择第一个不包含null值的唯一索引为主键索引
3. 如果没有唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐藏的聚集索引，它会随着行记录的写入而主键递增。

**所以，使用行锁的时候为什么锁表，是因为查询没有使用索引，会进行全表扫描，然后把每一个隐藏的聚集索引都锁住了。**

2、为什么通过唯一索引给数据行加锁，主键索引也会被锁住？

在辅助索引里面，索引存储的是二级索引和主键的值。比如name=4，存储的是name的索引和主键 id 的值 4。

而主键索引里面除了索引之外，还存储了完整的数据。所以我们通过辅助索引锁定一行数据的时候，它跟我们检索数据的步骤是一样的，会通过主键值找到主键索引，然后也锁定。



### InnoDB中的行锁锁住的范围

Record：数据库里面存在的主键值，我们把它叫做 Record（记录）

Gap：根据主键，这些存在的 Record 隔开的数据不存在的区间，我们把它叫做 Gap，间隙，它是一个左开右开的区间。

Next-key:最后一个，间隙（Gap）连同它左边的记录（Record），叫做临键的区间，它是一个左开右闭的区间。

Record lock:记录锁

当我们对于唯一性的索引（包括唯一索引和主键索引）使用等值查询，精准匹配到一条记录的时候，这个时候使用的就是记录锁。用不同的 key 去加锁，不会冲突，它只锁住这个 record。

Gap lock：间隙锁

当我们查询的记录不存在，没有命中任何一个 record，无论是用等值查询还是范围查询的时候，它使用的都是间隙锁。当查询的记录不存在的时候，使用间隙锁

Next Gap lock：临键锁

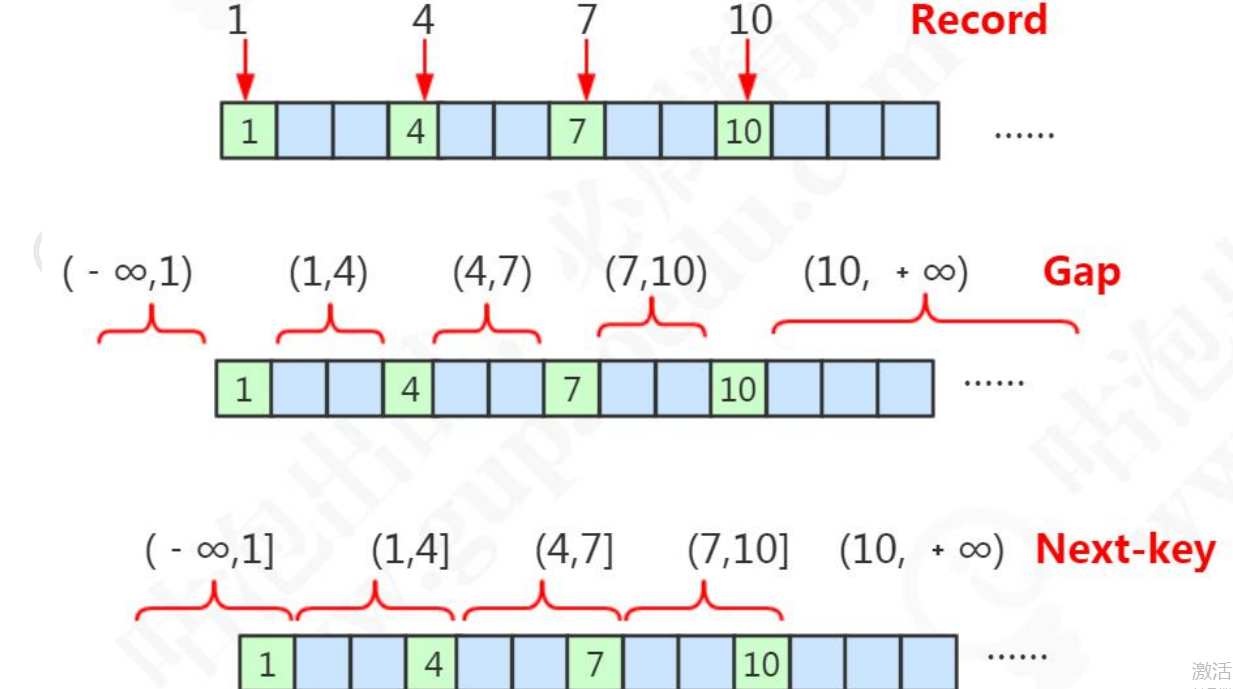
当我们使用了范围查询，不仅仅命中了 Record 记录，还包含了 Gap

间隙，在这种情况下我们使用的就是临键锁，它是 MySQL 里面默认的行锁算法，相当于记录锁加上间隙锁。

其他两种退化的情况：

唯一性索引，等值查询匹配到一条记录的时候，临键锁退化成记录锁。没有匹配到任何记录的时候，退化成间隙锁。

临键锁，锁住最后一个 key 的下一个左开右闭的区间。



select \* from t2 where id >5 and id <=7 for update; -- 锁住(4,7]和(7,10]

select \* from t2 where id >8 and id <=10 for update; -- 锁住 (7,10]，(10,+∞)

为什么要锁住下一个左开右闭的区间？——就是为了解决幻读的问题。

**隔离级别的实现**

未提交读（Read Uncommited）:RU 隔离级别：不加锁。

串行化（Serializable）：所有的 select 语句都会被隐式的转化为 select ... in share mode，会和 update、delete 互斥。

可重复读取（Repeatable Read）：RR 隔离级别下，普通的 select 使用快照读(snapshot read)**，底层使用 MVCC 来实现**。**加锁的** select(select ... in share mode / select ... for update)以及更新操作update, delete 等语句使用当前读（current read），**底层使用记录锁、或者间隙锁、临键锁。所以InnoDB的rr不会出现幻读**

读已提交（Read Commited）：

RC 隔离级别下，普通的 select 都是快照读，**使用 MVCC 实现**。加锁的 select 都使用记录锁，**因为没有 Gap Lock。（只用了记录锁）**

除了两种特殊情况——外键约束检查(foreign-key constraint checking)以及重复键检查(duplicate-key checking)时会使用间隙锁封锁区间。

**所以 RC 会出现幻读的问题。**

事务隔离级别的选择

RU 和 Serializable 肯定不能用

RC 和 RR 主要有几个区别：

1、 RR 的间隙锁会导致锁定范围的扩大。

2、 条件列未使用到索引，RR 锁表，RC 锁行。

3、 RC 的“半一致性”（semi-consistent）读可以增加 update 操作的并发性。

在 RC 中，一个 update 语句，如果读到一行已经加锁的记录，此时 InnoDB 返回记录最近提交的版本，由 MySQL 上层判断此版本是否满足 update 的 where 条件。若满足(需要更新)，则 MySQL 会重新发起一次读操作，此时会读取行的最新版本(并加锁)。

实际上，如果能够**正确地使用锁（避免不使用索引去枷锁），只锁定需要的数据，**用默认的 RR 级别就可以了。

### 死锁

排它锁有互斥的特性。一个事务或者说一个线程持有锁的时候，会阻止其他的线程获取锁，这个时候会造成阻塞等待，如果循环等待，会有可能造成死锁

#### 锁什么时候释放

事务结束（commit，rollback）和客户端连接断开。

如果一个事务一直未释放锁，其他事务会被阻塞多久？会不会永远等待下去？如果是，在并发访问比较高的情况下，如果大量事务因无法立即获得所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖跨数据库。

MySQL 有一个参数来控制获取锁的等待时间，默认是 50 秒。

show VARIABLES like 'innodb\_lock\_wait\_timeout';

#### 死锁的发生和检测

死锁的产生条件

因为锁本身是互斥的，（1）同一时刻只能有一个事务持有这把锁，（2）其他的事务需要在这个事务释放锁之后才能获取锁，而不可以强行剥夺，（3）当多个事务形成等待环路的时候，即发生死锁。

在发生死锁时，InnoDB 一般都能通过算法（wait-for graph）自动检测到。这样就不用等待50秒

#### 查看锁信息

show status like 'innodb\_row\_lock\_%';

Innodb\_row\_lock\_current\_waits：当前正在等待锁定的数量；

Innodb\_row\_lock\_time ：从系统启动到现在锁定的总时间长度，单位 ms；

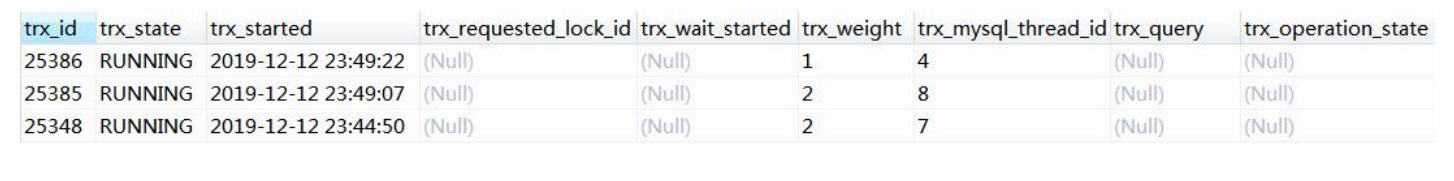
Innodb\_row\_lock\_time\_avg ：每次等待所花平均时间；

Innodb\_row\_lock\_time\_max：从系统启动到现在等待最长的一次所花的时间；

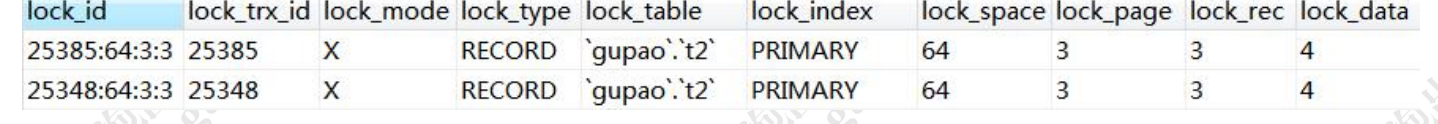
Innodb\_row\_lock\_waits ：从系统启动到现在总共等待的次数。

SHOW 命令是一个概要信息。InnoDB 还提供了三张表来分析事务与锁的情况：

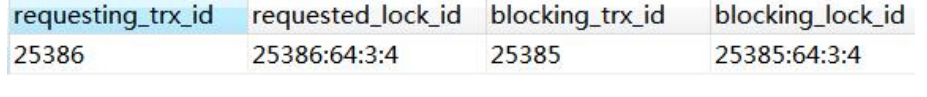
select \* from information\_schema.INNODB\_TRX; -- 当前运行的所有事务 ，还有具体的语句



select \* from information\_schema.INNODB\_LOCKS; -- 当前出现的锁



select \* from information\_schema.INNODB\_LOCK\_WAITS; -- 锁等待的对应关系



如果一个事务长时间持有锁不释放，可以 kill 事务对应的线程 ID，也就是

INNODB\_TRX 表中的 trx\_mysql\_thread\_id，例如执行 kill 4，kill 7，kill 8。

当然，死锁的问题不能每次都靠 kill 线程来解决，这是治标不治本的行为。我们应该尽量在应用端，也就是在编码的过程中避免。

#### 死锁的避免

1、 在程序中，操作多张表时，尽量以相同的顺序来访问（避免形成等待环路）；

2、 批量操作单张表数据的时候，先对数据进行排序（避免形成等待环路）；

3、 申请足够级别的锁，如果要操作数据，就申请排它锁；

4、 尽量使用索引访问数据，避免没有 where 条件的操作，避免锁表；

5、 如果可以，大事务化成小事务；

6、 使用等值查询而不是范围查询查询数据，命中记录，避免间隙锁对并发的影响。

# Mysql性能优化

大部分时候想要实现的目标是让我们的查询更快。一个查询的动作又是由很多个环节组成的，每个环节都会消耗时间，我们要减少查询所消耗的时间，就要从每一个环节入手。

## 连接配置优化

第一个环节是客户端连接到服务端，连接这一块有可能会出现什么样的性能问题？

有可能是服务端连接数不够导致应用程序获取不到连接。

从两个方面来解决连接数不够的问题：

1、从服务端来说，我们可以增加服务端的可用连接数。

如果有多个应用或者很多请求同时访问数据库，连接数不够的时候，可以：

（1）修改配置参数**增加可用连接数**，修改 max\_connections 的大小：

show variables like 'max\_connections'; -- 修改最大连接数，当有多个应用连接的时候

（2）**及时释放不活动的连接**。交互式和非交互式的客户端的默认超时时间都是 28800 秒，8 小时，我们可以把这个值调小。

show global variables like 'wait\_timeout'; --及时释放不活动的连接，注意不要释放连接池还在使用的连接

2、从客户端来说，可以减少从服务端获取的连接数，如果我们想要不是每一次执行

SQL 都创建一个新的连接，应该怎么做？这个时候我们可以引入连接池，实现连接的重用。

我们可以在哪些层面使用连接池？ORM 层面（MyBatis 自带了一个连接池）；或者使用专用的连接池工具（阿里的 Druid、Spring Boot 2.x 版本默认的连接池 Hikari、老牌的 DBCP 和 C3P0）。

当客户端改成从连接池获取连接之后，连接池的大小应该怎么设置呢？

大家可能会有一个误解，觉得连接池的最大连接数越大越好，这样在高并发的情况下客户端可以获取的连接数更多，不需要排队。

实际情况并不是这样。**连接池并不是越大越好**，只要维护一定数量大小的连接池，其他的客户端排队等待获取连接就可以了。有的时候连接池越大，效率反而越低。

在 Hikari 的 github 文档中，给出了一个 PostgreSQL 数据库建议的设置连接池大小的公式：https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing

它的建议是机器核数乘以 2 加 1。也就是说，4 核的机器，连接池维护 9 个连接就够了。这个公式从一定程度上来说对其他数据库也是适用的。这里面还有一个减少连接池大小实现提升并发度和吞吐量的案例。

每一个连接，服务端都需要创建一个线程去处理它。连接数越多，服务端创建的线程数就会越多。如果线程数量超过了服务器的cpu核心数，那么就会进行上下文切换，CPU 的核数是有限的，频繁的上下文切换会造成比较大的性能开销。

从数据库配置的层面去优化数据库。不管是数据库本身的配置，还是安装这个数据库服务的操作系统的配置，对于配置进行优化，最终的目标都是为了更好地发挥硬件本身的性能，包括 CPU、内存、磁盘、网络。

MySQL 和 InnoDB 的配置参数，包括各种开关和数值的配置，大多数参数都提供了一个默认值，比如默认的 buffer\_pool\_size，默认的页大小，InnoDB 并发线程数等等。这些默认配置可以满足大部分情况的需求，除非有特殊的需求，在清楚参数的含义的情况下再去修改它。修改配置的工作一般由专业的 DBA 完成。

## 缓存—架构优化

### 缓存

在应用系统的并发数非常大的情况下，如果没有缓存，会造成两个问题：一方面是会给数据库带来很大的压力。另一方面，从应用的层面来说，操作数据的速度也会受到影响。我们可以用第三方的缓存服务来解决这个问题，例如 Redis。

运行独立的缓存服务，属于架构层面的优化。

### 主从复制

如果单台数据库服务满足不了访问需求，那我们可以做数据库的集群方案

集群的话必然会面临一个问题，就是不同的节点之间数据一致性的问题。如果同时读写多台数据库节点，怎么让所有的节点数据保持一致？

这个时候我们需要用到复制技术（replication），被复制的节点称为 master，复制

的节点称为 slave。slave 本身也可以作为其他节点的数据来源，这个叫做级联复制。

主从复制是怎么实现的呢？

更新语句会记录 binlog，它是一种逻辑日志。有了这个 binlog，从服务器会获取主服务器的 binlog 文件，然后解析里面的 SQL语句，在从服务器上面执行一遍，保持主从的数据一致。

这里面涉及到三个线程，连接到 master 获取 binlog，并且解析 binlog 写入中继日志，这个线程叫做 **I/O 线程**。

Master 节点上有一个 **log dump 线程**，是用来发送 binlog 给 slave 的。

从库的 **SQL 线程**，是用来读取 relay log，把数据写入到数据库的。

做了主从复制的方案之后，我们只把数据写入 master 节点，而读的请求可以分担到

slave 节点。我们把这种方案叫做读写分离。读写分离可以一定程度低减轻数据库服务器的访问压力，但是需要特别注意主从数据一致性的问题。

基于主从复制的原理，我们需要弄明白，主从复制到底慢在哪里？

1. 单线程

在早期的 MySQL 中，**slave 的 SQL 线程是单线程**。master 可以支持 SQL 语句的并行执行，配置了多少的最大连接数就是最多同时多少个 SQL 并行执行。而 slave 的 SQL 却只能单线程排队执行，在主库并发量很大的情况下，同步数据肯定会出现延迟。

为什么从库上的 SQL Thread 不能并行执行呢？举个例子，主库执行了多条 SQL 语

句，首先用户发表了一条评论，然后修改了内容，最后把这条评论删除了。这三条语句在**从库上的执行顺序肯定是不能颠倒的**。

insert into user\_comments (10000009,'nice');

update user\_comments set content ='very good' where id =10000009;

delete from user\_comments where id =10000009;

怎么解决这个问题呢？怎么减少主从复制的延迟？

1. 异步与全同步

在主从复制的过程中，MySQL 默认是异步复制的。也就是说，对于主节点来说，写入 binlog，事务结束，就返回给客户端了。对于 slave 来说，接收到 binlog，就完事儿了，master 不关心 slave 的数据有没有写入成功。

如果要减少延迟，是不是可以**等待全部从库的事务执行完毕，才返回给客户端呢**？这样的方式叫做全同步复制。**从库写完数据，主库才返会给客户端**。

这种方式虽然可以保证在读之前，数据已经同步成功了，但是带来的副作用大家应该能想到，**事务执行的时间会变长**，它会导致 master 节点性能下降。有没有更好的办法呢？既减少 slave 写入的延迟，又不会明显增加 master 返回给客户端的时间？

1. 半同步复制

主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到 binlog 并写到 relay log 中才返回给客户端。master 不会等待很长的时间，但是返回给客户端的时候，数据就即将写入成功了，因为它只剩最后一步了：就是读取 relay log，写入从库。

半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，它需要等待一个 slave 写入中继日志，这里多了一个网络交互的过程，所以，半同步复制最好在低延时的网络中使用。

1. 多库并行复制

如果 3 条语句是在三个数据库执行，操作各自的数据库，是不是肯定不会产生并发的问题呢？执行的顺序也没有要求。当然是，所以如果是操作三个数据库，这三个数据库的从库的 SQL 线程可以并发执行。这是 MySQL 5.6版本里面支持的多库并行复制。

但是在大部分的情况下，我们都是单库多表的情况，在一个数据库里面怎么实现并行复制呢？或者说，我们知道，数据库本身就是支持多个事务同时操作的；为什么这些事务在主库上面可以并行执行，却不会出现问题呢？

因为他们本身就是互相不干扰的，比如这些事务是操作不同的表，或者操作不同的行，不存在资源的竞争和数据的干扰。那在主库上并行执行的事务，在从库上肯定也是可以并行执行，是不是？比如在 master 上有三个事务同时分别操作三张表，这三个事务是不是在 slave 上面也可以并行执行呢？

1. 异步复制之 GTID 复制

可以把那些在主库上并行执行的事务，分为一个组，并且给他们编号，这一个组的事务在从库上面也可以并行执行。这个编号，我们把它叫做 GTID（GlobalTransaction Identifiers），这种主从复制的方式，我们把它叫做基于 GTID 的复制。

如果我们要使用 GTID 复制，我们可以通过修改配置参数打开它，默认是关闭的：show global variables like 'gtid\_mode';无论是优化 master 和 slave 的连接方式，还是让从库可以并行执行 SQL，都是从数据库的层面去解决主从复制延迟的问题。

### 分库分表

如果单个 master 节点或者单张表存储的数据过大的时候，比如一张表有上亿的数据，单表的查询性能还是会下降，我们要进一步对单台数据库节点的数据分型拆分，这个就是分库分表。

**垂直分库**，减少并发压力。就是根据不同的业务进行拆分的，拆分成不同的数据库，不同的业务对应不同数据库，比如会员数据库、订单数据库、支付数据库、消息数据库等，垂直拆分在大型电商项目中使用比较常见。

**优点**：拆分后业务清晰，拆分规则明确，系统之间整合或扩展更加容易。

**缺点**：部分业务表无法join，跨数据库查询比较繁琐(必须通过接口形式通讯(http+json))、会产生分布式事务的问题,提高了系统的复杂度。举栗子：不可能出现，在订单服务中，订单服务直接连接会员服务的数据库这种情况。

**水平分表**，解决存储瓶颈，把同一张表中的数据拆分到不同的数据库中进行存储、或者把一张表拆分成 n 多张小表。水平拆分不是将表的数据做分类，而是按照某个字段的某种规则来分散到多个库之中，每个表中包含一部分数据

### 高可用方案

通过主从或者分库分表可以减少单个数据库节点的访问压力和存储压力，达到提升数据库性能的目的，但是如果 master 节点挂了，怎么办？所以，高可用（High Available）也是高性能的基础。

主从复制

传统的 HAProxy + keepalived 的方案，基于主从复制。

NDB Cluster

基于 NDB 集群存储引擎的 MySQL Cluster。

### 总结

高可用 HA 方案需要解决的问题都是当一个 master 节点宕机的时候，如何提升一个数据最新的 slave 成为 master。如果同时运行多个 master，又必须要解决 master 之间数据复制，以及对于客户端来说连接路由的问题。不同的方案，实施难度不一样，运维管理的成本也不一样。以上是**架构层面的优化，可以用缓存，主从，分库分表**。

## 解析器

解析器，词法和语法分析，主要保证语句的正确性，语句不出错就没问题。由 Sever

自己处理，跳过。

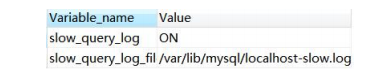
## 优化器——SQL 语句分析与优化

优化器就是对我们的 SQL 语句进行分析，生成执行计划。

把 SQL 执行情况记录下来

### 慢查询日志 slow query log

因为开启慢查询日志是有代价的（跟 bin log、optimizer-trace 一样），所以它默认是关闭的： show variables like 'slow\_query%';



除了这个开关，还有一个参数，控制执行超过多长时间的 SQL 才记录到慢日志，默认是 10 秒。可以直接动态修改参数（重启后失效）。

set @@global.slow\_query\_log=1; -- 1 开启，0 关闭，重启后失效

set @@global.long\_query\_time=3; -- mysql 默认的慢查询时间是 10 秒，另开一个窗口后才会查到最新值

show variables like '%long\_query%';

show variables like '%slow\_query%';

或者修改配置文件 my.cnf。

以下配置定义了慢查询日志的开关、慢查询的时间、日志文件的存放路径。

slow\_query\_log = ON

long\_query\_time=2

slow\_query\_log\_file =/var/lib/mysql/localhost-slow.log

模拟慢查询：

select sleep(10);

查询 user\_innodb 表的 500 万数据（检查是不是没有索引）。

SELECT \* FROM `user\_innodb` where phone = '136';

### 慢日志分析

1、日志内容

show global status like 'slow\_queries'; -- 查看有多少慢查询

show variables like '%slow\_query%'; -- 获取慢日志目录

2、mysqldumpslow

MySQL 提供了 mysqldumpslow 的工具，在 MySQL 的 bin 目录下。

mysqldumpslow --help

例如：查询用时最多的 20 条慢 SQL：

mysqldumpslow -s t -t 20 -g 'select' /var/lib/mysql/localhost-slow.log

Count 代表这个 SQL 执行了多少次；

Time 代表执行的时间，括号里面是累计时间；

Lock 表示锁定的时间，括号是累计；

Rows 表示返回的记录数，括号是累计。

除了慢查询日志之外，还有一个 SHOW PROFILE 工具可以使用。

SHOW PROFILE

SHOW PROFILE 是谷歌高级架构师 Jeremy Cole 贡献给 MySQL 社区的，可以查看 SQL 语句执行的时候使用的资源，比如 CPU、IO 的消耗情况。

在 SQL 中输入 help profile 可以得到详细的帮助信息。

查看是否开启

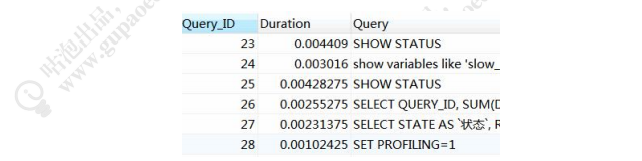
select @@profiling;

set @@profiling=1;

查看 profile 统计

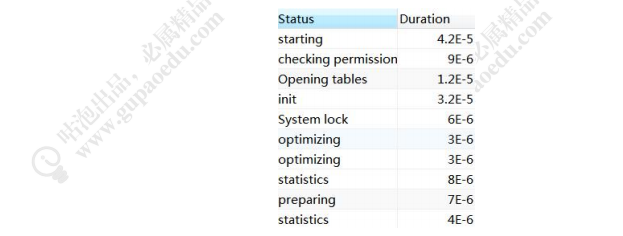
（命令最后带一个 s）

show profiles;



查看最后一个 SQL 的执行详细信息，从中找出耗时较多的环节（没有 s）。

show profile;



6.2E-5，小数点左移 5 位，代表 0.000062 秒。

也可以根据 ID 查看执行详细信息，在后面带上 for query + ID。

show profile for query 1;

除了慢日志和 show profile，如果要分析出当前数据库中执行的慢的 SQL，还可以

通过查看运行线程状态和服务器运行信息、存储引擎信息来分析。

其他系统命令

show processlist 运行线程

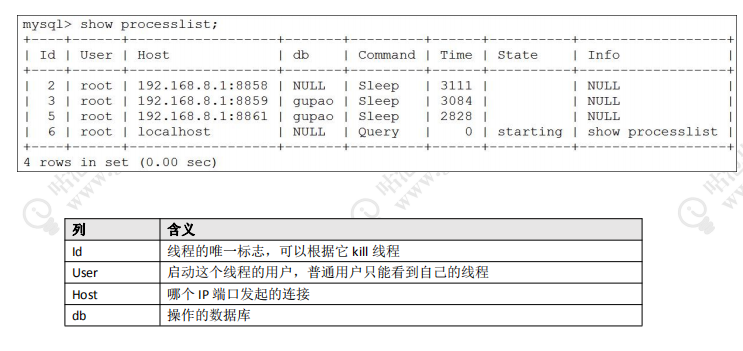
https://dev.mysql.com/doc/refman/5.7/en/show-processlist.html

show processlist;

这是很重要的一个命令，用于显示用户运行线程。可以根据 id 号 kill 线程。

也可以查表，效果一样：

select \* from information\_schema.processlist;





show status 服务器运行状态

https://dev.mysql.com/doc/refman/5.7/en/show-status.html

SHOW STATUS 用于查看 MySQL 服务器运行状态（重启后会清空），有 session

和 global 两种作用域，格式：参数-值。

可以用 like 带通配符过滤。

SHOW GLOBAL STATUS LIKE 'com\_select'; -- 查看 select 次数

show engine 存储引擎运行信息

https://dev.mysql.com/doc/refman/5.7/en/show-engine.html

show engine 用来显示存储引擎的当前运行信息，包括事务持有的表锁、行锁信息；

事务的锁等待情况；线程信号量等待；文件 IO 请求；buffer pool 统计信息。

例如：

show engine innodb status;

如果需要将监控信息输出到错误信息 error log 中（15 秒钟一次），可以开启输出。

show variables like 'innodb\_status\_output%';

-- 开启输出：

SET GLOBAL innodb\_status\_output=ON;

SET GLOBAL innodb\_status\_output\_locks=ON;

我们现在已经知道了这么多分析服务器状态、存储引擎状态、线程运行信息的命令，

如果让你去写一个数据库监控系统，你会怎么做？

其实很多开源的慢查询日志监控工具，他们的原理其实也都是读取的系统的变量和

状态。

现在我们已经知道哪些 SQL 慢了，为什么慢呢？慢在哪里？

MySQL 提供了一个执行计划的工具（在架构中我们有讲到，优化器最终生成的就是

一个执行计划），其他数据库，例如 Oracle 也有类似的功能。

通过 EXPLAIN 我们可以模拟优化器执行 SQL 查询语句的过程，来知道 MySQL 是

怎么处理一条 SQL 语句的。通过这种方式我们可以分析语句或者表的性能瓶颈。

explain 可以分析 update、delete、insert 么？

MySQL 5.6.3以前只能分析 SELECT; MySQL5.6.3以后就可以分析update、delete、

insert 了。

### 执行计划

使用 **explain** 可以模拟优化器执行SQL语句，从而知道MySQL是何处理你的SQL语句的，分析你的查询语句或是结构的性能瓶颈。

通过explain，能够很清晰的知道SQL查询读取表的顺序、哪些索引被使用到、表直接的引用关系、每张表有多少条数据被扫描等等。

参数说明：

1. **id：** 表示select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序。

id相同，执行顺序由上至下；

id不同，如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行；

id相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行。

1. **select\_type：** 主要是用于区分普通查询、联合查询、子查询等。

**SIMPLE：** 简单的select查询，查询中不包含子查询或者UNION；

**PRIMARY：** 查询中若包含任何复杂的子部分，最外层查询则被标记为primary；

**SUBQUERY：** 在SELECT或者WHERE列表中包含了子查询；

**DERIVED：** 在FROM列表中包含的子查询被标记为DERIVED（衍生） MySQL会递归执行这些子查询，把结果放在临时表里；

**UNION：** 若第二个SELECT出现在UNION之后，则被标记为UNION; 若UNION包含在FROM子句的子查询中，外层SELECT将被标记为DERIVED；

**UNION RESULT：** 从UNION表获取结果的SELECT。

1. **table：** 查询涉及到的表，直接显示表名或者表的别名。

<unionM,N> 由ID为M,N 查询union产生的结果；

<subqueryN> 由ID为N查询生产的结果。

1. **type：** 访问类型，sql 查询优化中一个很重要的指标，结果值从好到坏依次是：system > const > eq\_ref > ref > range > index > ALL。

**system：** 表只有一行记录（等于系统表），const类型的特例，基本不会出现，可以忽略不计；

**const：** 表示通过索引一次就找到了，const用于比较primary key 或者 unique索引；

**eq\_ref：** 唯一索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描；

**ref：** 非唯一性索引扫描，返回匹配某个单独值的所有行，本质是也是一种索引访问；

**range：** 只检索给定范围的行，使用一个索引来选择行。key列显示使用了哪个索引一般就是在你的where语句中出现了between、<、>、in等的查询 这种范围扫描索引扫描比全表扫描要好，因为他只需要开始索引的某一点，而结束语另一点，不用扫描全部索引。

**index：** Full Index Scan，索引全表扫描，把索引从头到尾扫一遍；

**ALL：** Full Table Scan，遍历全表以找到匹配的行。

1. **possible\_keys：** 查询涉及的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用,也就是可能使用到的索引。
2. **key：** 实际使用的索引。如果为null则没有使用索引，查询中若使用了覆盖索引，则索引和查询的select字段重叠。
3. **key\_len：** 表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精确性的情况下，长度越短越好key\_len显示的值为索引最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的。
4. **ref：** 显示索引那一列被使用了，如果可能的话，是一个常数。那些列或常量被用于查找索引列上的值。
5. **rows：** 据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数。
6. **Extra：** 包含不适合在其他列中显示但十分重要的额外信息。

**Using filesort：** 说明mysql会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。 MySQL中无法利用索引完成排序操作成为“文件排序”；

**Using temporary：** 使用了临时表保存中间结果，MySQL在对查询结果排序时使用临时表。常见于排序order by 和分组查询 group by；

**Using index：** 表示相应的select操作中使用了覆盖索引（Coveing Index）,避免访问了表的数据行，效率不错！ 如果同时出现using where，表明索引被用来执行索引键值的查找； 如果没有同时出现using where，表面索引用来读取数据而非执行查找动作,覆盖索引（Covering Index） ；

**Using where：** 表明用到where条件过滤；

**Using join buffer：** 使用了连接缓存；

**Impossible where：** 子句的值总是false，不能用来获取任何元组；

**Select tables optimized away：** 在没有GROUPBY子句的情况下，基于索引优化MIN/MAX操作或者对于MyISAM存储引擎优化COUNT(\*)操作，不必等到执行阶段再进行计算， 查询执行计划生成的阶段即完成优化；

**distinct：** 优化distinct，在找到第一匹配的元组后即停止找同样值的工作。

1. **partitions：** 查询记录来自哪个分区。若没有匹配分区，该值为NULL。
2. **filtered：** 查询过滤行所占百分比，若为100则数据未过滤，过滤掉的行数为：总行数×filtered百分比值（单位%）。

## SQL 与索引优化

sql优化

1.查询语句中尽量不适用 \*

2.尽量减少子查询，使用关联查询（left join，right join，inner join）替代

3.减少使用IN或者NOT IN，使用exists，not exists或者关联查询语句替代

4.使用union或者union all代替or查询（确认没有重复数据或者不用剔除重复数据时，union all性能更好，解释器不用再扫描一遍表挑选重复）

5.建表的时候能使用数字类型字段就不要使用字符串，数字类型的字段作为查询条件比字符串快。

6.那些能过滤掉最大数量记录的条件必须写在where子句的最末尾。

7.查询字段很多的情况下，慎用distinct关键字，会大大降低查询效率。

8.定义字段时，尽量使用varchar而不是char。定义一个char[10]和varchar[10],如果存进去的是‘abcd’,

那么char所占的长度依然为10，除了字符‘abcd’外，

## 存储引擎

### 存储引擎的选择

为不同的业务表选择不同的存储引擎，例如：查询插入操作多的业务表，用 MyISAM。

临时数据用 Memeroy。常规的并发大更新多的表用 InnoDB。

### 分区或者分表

分区不推荐。

交易历史表：在年底为下一年度建立 12 个分区，每个月一个分区。

渠道交易表：分成当日表；当月表；历史表，历史表再做分区。

### 字段定义

原则：使用可以正确存储数据的最小数据类型。

为每一列选择合适的字段类型：

**整数类型**



INT 有 8 种类型，不同的类型的最大存储范围是不一样的。性别？用 TINYINT，因为 ENUM 也是整型存储。

**字符类型**

变长情况下，varchar 更节省空间，但是对于 varchar 字段，需要一个字节来记录长

度。

固定长度的用 char，不要用 varchar。

非空

非空字段尽量定义成 NOT NULL，提供默认值，或者使用特殊值、空串代替 null。

NULL 类型的存储、优化、使用都会存在问题。

不要用外键、触发器、视图

降低了可读性；

影响数据库性能，应该把把计算的事情交给程序，数据库专心做存储；

数据的完整性应该在程序中检查。

**大文件存储**

不要用数据库存储图片（比如 base64 编码）或者大文件；

把文件放在 NAS 上，数据库只需要存储 URI（相对路径），在应用中配置 NAS 服

务器地址。

### 表拆分

将不常用的字段拆分出去，避免列数过多和数据量过大。

比如在业务系统中，要记录所有接收和发送的消息，这个消息是 XML 格式的，用

blob 或者 text 存储，用来追踪和判断重复，可以建立一张表专门用来存储报文。

## 总结：优化体系



除了对于代码、SQL 语句、表定义、架构、配置优化之外，业务层面的优化也不能

忽视。举几个例子：

1）在某一年的双十一，为什么会做一个充值到余额宝和余额有奖金的活动（充 300

送 50）？

因为使用余额或者余额宝付款是记录本地或者内部数据库，而使用银行卡付款，需

要调用接口，操作内部数据库肯定更快。

2）在去年的双十一，为什么在凌晨禁止查询今天之外的账单？

这是一种降级措施，用来保证当前最核心的业务。

3）最近几年的双十一，为什么提前一个多星期就已经有双十一当天的价格了？

预售分流。

在应用层面同样有很多其他的方案来优化，达到尽量减轻数据库的压力的目的，比如限流，或者引入 MQ 削峰，等等等等。

为什么同样用 MySQL，有的公司可以扛住百万千万级别的并发，而有的公司几百个

并发都扛不住，关键在于怎么用。所以，用数据库慢，不代表数据库本身慢，有的时候

还要往上层去优化

当然，如果关系型数据库解决不了的问题，我们可能需要用到搜索引擎或者大数据的方案了，并不是所有的数据都要放到关系型数据库存储。

# 数据库表设计

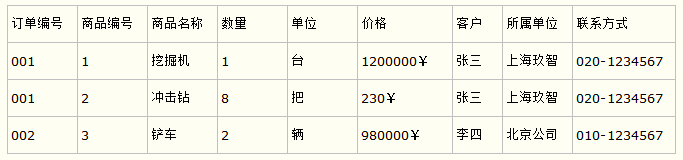
## 第一大范式

数据库表中的所有字段都只是具有单元属性；第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值

## 第二大范式

第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键，如下表所示。



这样就产生一个问题：这个表中是以订单编号和商品编号作为联合主键。这样在该表中商品名称、单位、商品价格等信息不与该表的主键相关，而仅仅是与商品编号相关。所以在这里违反了第二范式的设计原则。

而如果把这个订单信息表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中，就非常完美了。如下所示。



这样设计，在很大程度上减小了数据库表的冗余。如果要获取订单的商品信息，使用商品编号到商品信息表中查询即可。

## 第三大范式

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表。



这样在查询订单信息的时候，就可以使用客户编号来引用客户信息表中的记录，也不必在订单信息表中多次输入客户信息的内容，减小了数据冗余。

### 反范式化设计

范式化优点：可以尽量减少数据冗余，更新操作比反范式化要快，表比反范式化更小

范式化缺点：对于查询需要对多个表进行关联，虽然提高了写操作性能，但是降低了读的性能，在mysql关联成本很高的，就算再简单的查询也要关联一次表。我们需要对表进行适当的反范式化，以减小关联的数量

并且使我们更难对查询进行索引优化。例如将经常使用的列分在不同的表中，这些列在同一个表中可以属于同一个索引。不同表中就无法放到同一个索引中了。

反范式化时针对范式化而言的，为了性能和读取效率适当违反数据库设计范式，允许存在少量的冗余字段，也就是用空间来换时间

反范式的优点：可以减少表的关联，可以更好的进行索引优化

反范式的缺点：存在数据冗余以及数据修改的工作增加，